ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Fall 2013

Dan Negrut
Associate Professor
Department of Mechanical Engineering
University of Wisconsin, Madison
September 4, 2013

“I think there is a world market for maybe five computers.”
T. J. Watson, chairman of IBM, 1943.
● Purpose of today’s lecture
  ● Get a 30,000 perspective on this class and understand whether this is a class worth taking

● What we will cover today
  ● Course logistics
  ● Brief overview of syllabus
  ● Motivation and central themes of this class
  ● Start quick overview of C programming language
Instructor: Dan Negrut

- Polytechnic Institute of Bucharest, Romania
  - B.S. – Aerospace Engineering (1992)

- University of Iowa
  - Ph.D. – Mechanical Engineering (1998)

- MSC.Software
  - Product Development Engineer 1998-2005

- University of Michigan
  - Adjunct Assistant Professor, Dept. of Mathematics (2004)

- Division of Mathematics and Computer Science, Argonne National Laboratory

- University of Wisconsin-Madison, Joined in Nov. 2005
  - Research Focus: Computational Dynamics (Dynamics of Multi-body Systems)
  - Established the Simulation-Based Engineering Lab (http://sbel.wisc.edu)
Acknowledgements

- Students helping with this class
  - Ang Li [grader]
  - Andrew Seidl [takes care of hardware and software]
  - Hammad Mazhar [help with CUDA & thrust]

- NVIDIA, AMD & US Army ARO:
  - Financial support to build Euler, CPU/GPU cluster used in this class
Good to know…

- **Time**: 8:00-9:15 AM Mo & Wd & Fr
- **Location**: 2121ME
- **Office**: 2035ME
- **Phone**: 608 890-0914
- **E-Mail**: negrut@engr.wisc.edu
- **Course Webpage**: [http://sbel.wisc.edu/Courses/ME964/2013/index.htm](http://sbel.wisc.edu/Courses/ME964/2013/index.htm)
- **Grades reported at**: [learnuw.wisc.edu](http://learnuw.wisc.edu)
Office Hours:
- Monday 2 – 3:30 PM
- Friday 2 – 3:30 PM

Call or email to arrange for meetings outside office hours

Walk-ins are fine as long as they are in the afternoon
References

- No textbook is required, but there are some recommended ones:

  - **Highly recommended**
    - NVIDIA CUDA C Programming Guide V5.5, 2013
    - Jason Sanders and Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010 (on reserve, Wendt Lib.)
    - Peter Pacheco: An Introduction to Parallel Programming, Morgan Kaufmann, 2011
    - B. Kernighan and D. Ritchie, The C Programming Language
    - B. Stroustrup, The C++ Programming Language, Third Edition
Further reading

- D. Negrut, Primer: Elements of Processor Architecture. The Hardware/Software Interplay, available on class website
- Wen-mei W. Hwu (editor), GPU Gems 4, 2011, Addison Wesley
- Rob Farber: CUDA Application Design and Development, Morgan Kaufmann 2011
- H. Nguyen (editor), GPU Gems 3, Addison Wesley, 2007 (on reserve, Wendt Lib.)
- Peter Pacheco: Parallel Programming with MPI, Morgan Kaufmann, 1996
- Michael J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2003
Course Related Information

- This course is offered on an accelerated track
- Three lectures per week, each 75 minutes long
- Last lecture: November 8
  - 29 lectures total, just like a regular semester yet compressed in two months
- No class after November 8
  - I will still have office hours
  - Homework will continue to be assigned on a weekly schedule past Nov. 8
- Motivation:
  - It’ll give us more than one month to work on a meaningful Final Project
    - More on this later
Course Related Information

- Handouts will be printed out and provided before each lecture
- Lecture material (PDF and audio) will be made available online at class website
- Looking into retrieving videos of this class (Spring 2012 edition)
- Grades will be maintained online at Learn@UW
- Syllabus will be updated as we go
  - It will contain info about
    - Topics we cover
    - Homework assignments
  - Available at the course website
    - [http://sbel.wisc.edu/Courses/ME964/2013/](http://sbel.wisc.edu/Courses/ME964/2013/)
The 964 Issue

- Class first taught in 2008
- Called ME964
- 900-level classes are experimental, need to change to 700 format
- Now cross-listed in ME, ECE, EMA, and CS as 759
- All old websites, links, forum, etc. – still reference the 964 number
- Apologies for any confusion this might cause
Grading

- Homework 40%
- Midterm Exam 15%
- Midterm Project 15%
- Final Project 25%
- Course Participation 5%

- Total 100%

NOTE:
- Score related questions (homework/exam) must be raised prior to next class after the homework/exam is returned.
Homework Policies

- There will be 12 HWs assigned
  - No late HW accepted
    - HW due at 11:59 PM on the due day

- The assignments with two lowest scores will be dropped when computing final score

- Homework and projects should be handed in using Learn@UW dropbox
  - There will be a window when you can submit your homework

- This class is hard because of the assignments. Very time consuming
Midterm Exam

- One midterm exam only, accounts for 15% of final grade

- Scheduled during regular class hours

- Tentatively scheduled on **November 8**
  - Review offered the day before, time/location TBA

- Doesn’t require use of a computer (it’s a pen and paper exam)

- It’s a “closed books” exam

- Covers the entire material discussed in class up to that point
Has to do with implementation of a parallel solution for solving a large *dense* banded system of equations
- Size: as high as you can go
- Implemented in CUDA
- Focus on banded matrices

Due on **November 15** at 11:59 PM

Accounts for 15% of final grade

Project is individual or produced by two-student teams

Should contain a comparison of your parallel code with solvers that are available already in the Scientific Computing community
- Intel MKL, LAPACK, Pardiso, etc,

Should include profiling results and a weak scaling analysis
Final Exam Project

- There will be no final exam but rather a Final Project

- The Final Project is due on at 11:59 PM on the Monday of the finals week

- Each student/team will present the project in a 30 minute time slot

- Presentation time slots will be posted in doodle for you to choose a convenient one
Final Exam Project

- Final Project (accounts for 25% of final grade):
  - It is an individual project or produced by a two-student team
  - You choose a problem that suites your research or interests
  - You are encouraged to tackle a meaningful problem
    - Attempt to solve a useful problem rather than a problem that you are confident that you can solve
    - Projects that are not successful are ok, provided you aim high enough and demonstrate good work
    - Continuing the Midterm Project topic is ok (shifting focus on sparse systems)
  - Work on Final Project starts on Nov. 15 after submitting project proposal
Class Participation

- Accounts for 5% of final grade. To earn the 5%, you must:
  - Contribute at least five meaningful posts on the class Forum
    - Forum is live at: http://sbel.wisc.edu/Forum/index.php?board=15.0
    - Forum meant to serve as a quick way to answer some of your questions by instructor and other 759 colleagues
    - You should get an email with login info shortly (today or tomorrow)
Scores and Grades

<table>
<thead>
<tr>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>92-100</td>
<td>A</td>
</tr>
<tr>
<td>86-91</td>
<td>AB</td>
</tr>
<tr>
<td>78-85</td>
<td>B</td>
</tr>
<tr>
<td>70-77</td>
<td>BC</td>
</tr>
<tr>
<td>60-69</td>
<td>C</td>
</tr>
<tr>
<td>50-59</td>
<td>D</td>
</tr>
</tbody>
</table>

- Grading will not be done on a curve
- Final score will be rounded to the nearest integer prior to having a letter assigned
  - Example:
    - 85.59 becomes AB
    - 85.27 becomes B
Prerequisites

- This is a high-level graduate class in a very fluid topic

- Familiarity with C is needed
  - You can probably be fine if you are a friend of Java

- Decent programming skills are necessary
  - Understanding pointers
  - Being able to wrestle with a compile error on your own
  - Having used a debugger
  - Having used a profiler
Rules of Engagement

- You are encouraged to discuss assignments with other class students
  - Post and read posts on Forum

- Getting **verbal** advice and suggestions from anybody is fine

- copy/paste of non-trivial code is not acceptable
  - Non-trivial = more than a line or so
  - Includes reading someone else’s code and then going off to write your own

- Use of third party libraries that directly implement the solution of a HW/Project is not acceptable unless explicitly asked to do so
A Word on Hardware…

- The course designed to leverage a dedicated CPU/GPU cluster
  - Called Euler

- Each student receives an individual account that will be used for
  - GPU computing
  - MPI-enabled parallel computing
  - OpenMP multi-core computing

- Advice: if possible, do all the programming on a local machine. Move to Euler for “production” runs
ME759
Heterogeneous Cluster
ME759  
Heterogeneous Cluster

- More than 50,000 GPU scalar processors
- More than 1,200 CPU cores
- Fast Mellanox Infiniband Interconnect (QDR), 40Gb/sec
- About 2.7 TB of RAM
- More than 20 Tflops Double Precision
A Word on Software…

● We will use Linux as our operating system of choice
  ● Euler runs Linux

● We’ll use the following versions of libraries/releases:
  ● CUDA: 5.0
  ● MPI: 2.0
  ● OpenMP: 3.0

● Reliance on makefiles generated with CMake, a build utility tool
  ● Scripts will be available to you in order to facilitate compile/link/debug/profile process

● We will use a suite of debugging and profiling tools
  ● gdb: debugger under Linux
  ● cuda-gdb: debugger for CUDA applications running on the GPU
  ● NVIDIA Profiler: Nsight

● Most of these tools are embedded in Eclipse
  ● OK to work under Windows, yet make sure your code compiles/runs on Euler before submitting
Staying in Touch...

- Please do not email me unless you have a personal problem
  - Examples:
    - Good: Schedule a one-on-one meeting outside office hours
    - Bad: Asking me clarifications on Problem 2 of the current assignment (this needs to be on the Forum)
    - Bad: telling me that you can’t compile your code (this should also go to the Forum)

- Any course-related question should be posted on the Forum
  - I continuously monitor the Forum
  - If you can answer a Forum post, please do so (counts towards your 5% class participation and helps me as well)
  - Keeps all of us on the same page

- The forum is *very* useful
Course Emphasis

- There are multiple choices when it comes to implementing parallelism
  - PThreads, Intel’s TBB, OpenMP, MPI, Ct, Cilk, CUDA, etc.

- Course focuses on parallelism enabled by
  - The Graphics Processing Unit (GPU), mostly aimed at fine grain level parallelism
  - OpenMP standard, aimed both at fine and coarse level parallelism
  - Message Passing Interface (MPI) standard, aimed at coarse grain parallelism

- This is not going to be a hard course but it’ll be a very busy course
  - You’ll easily understand all the material that we’ll cover (no rocket science)
  - The assignments are going to be time consuming
    - Writing software is time consuming
    - Writing parallel computing software adds insult to injury
Course Objectives

- Get familiar with today’s High-Performance Computing (HPC) software and hardware
  - Usually “high-performance” implies execution on parallel architectures; i.e., architectures that have the potential to finish a run much faster than when the same application is executed sequentially

- Help you recognize applications/problems that can draw on HPC

- Help you gain basic skills that will help you map these applications onto a parallel computing hardware/software stack
  - Write code, build, link, run, debug, profile

- Introduce basic software design patterns for parallel computing
Course Objectives
[Cntd.]

- **What I’ll try to accomplish**
  - Provide enough information for you to start writing software that can leverage parallel computing to hopefully reduce the amount of time required by your simulations to complete

- **What I will not attempt to do**
  - Investigate how to design new parallel computing languages or language features, compilers, how new hardware should be designed, etc.

- **To summarize,**
  - I’m a Mechanical Engineer, a consumer of parallel computing
  - Focus is not on how to design parallel computing hardware or instruction architecture sets for parallel computing
High Performance Computing for Engineering Applications

Why This Title?

- Computer Science: ISA, Limits to Instruction Level Parallelism and Multithreading, Speculative Execution, Pipelining, Memory Hierarchy, Memory Models, Cache Coherence, etc.
  - Long story short: how should a processor be built?

- Electrical Engineering: how will we build the processor that the CS colleagues have in mind?
  - Lots of microarchitecture issues

- This class: how to use the system built by electrical engineers who implemented the architecture devised by the CS colleagues
  - At the end of the day, in our research in Science/Engineering we'll be dealing with one of the seven dwarfs…
Phillip Colella’s “Seven Dwarfs”

High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- If add four more for embedded, covers all 41 EEMBC benchmarks
  8. Search/Sort
  9. Filter
  10. Combinational logic
  11. Finite State Machine
Profiling:
Who Will Be the Typical 759 Student?

- 37 students enrolled coming from four UW departments
  - Computer Science, Electrical Engineering, Engineering Mechanics, Mechanical Engineering

- “High Performance Computing for Engineering Applications”
  - There is no need to have a prior Engineering degree
  - The course assumes a level of programming experience of a typical Engineer
Auditing the Course

- **Why auditing?**
  - Augments your experience with this class
    - You get an account on the CPU/GPU cluster
    - You will be added to the email list
    - Can post questions on the forum

- **How to register for auditing:**
  - In order to audit a course, a student must first enroll in the course as usual. Then the student must request to audit the course online. (There is a tutorial available through the Office of the Registrar.) Finally, the student must save & print the form. Once they have obtained the necessary signatures, the form should be turned in to the Academic Dean in the Grad School at 217 Bascom. The Grad School offers more information on Auditing Courses in their Academic Policies and Procedures.

Tutorial website: [http://www.registrar.wisc.edu/isis_helpdocs/enrollment_demos/V90CourseChangeRequest/V90CourseChangeRequest.htm](http://www.registrar.wisc.edu/isis_helpdocs/enrollment_demos/V90CourseChangeRequest/V90CourseChangeRequest.htm)
Auditing Courses: [http://www.grad.wisc.edu/education/acadpolicy/guidelines.html#13](http://www.grad.wisc.edu/education/acadpolicy/guidelines.html#13)
Overview of Material Covered

[Fall 2013]

- Quick C Intro
- General considerations in relation to trends in the chip industry
- Overview of parallel computation paradigms and supporting hardware/software
- GPU computing and the CUDA programming model
- GPU parallel computing using the thrust template library
- OpenMP programming
- MPI programming
At the beginning of the road…

- Teaching the class for the fourth time
  - Rough edges remain
  - There might be questions that I don’t have an answer for
    - I’ll follow up on these and get back with you (on the Forum)

- Please ask questions (be curious)
My Advice to You [is simple]

- If you can, innovate, do something remarkable, amaze the rest of us…
End ME759 Overview

Beginning: Quick Review of C

- Essential reading: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
- Acknowledgement: Slides on this C Intro include material due to Donghui Zhang and Lewis Girod
- If these things look unfamiliar, please read the Primer document available on the class website
C Syntax and Hello World

#include <stdio.h>  // Header files
#include <stdio.h>  // Header files
#include <stdio.h>  // Header files
#include <stdio.h>  // Header files

/* The simplest C Program */
/* The simplest C Program */
/* The simplest C Program */
/* The simplest C Program */

int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

What do the <> mean?

#include inserts another file. "h" files are called "header" files. They contain declarations/definitions needed to interface to libraries and code in other "c" files.

A comment, ignored by the compiler

Return '0' from this function

The main() function is always where your program starts running.

Blocks of code ("lexical scopes") are marked by { ... }

#include <stdio.h>  // Header files
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
A Quick Digression About the Compiler

Compilation occurs in two steps: “Preprocessing” and “Compiling”

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \ )

The compiler then converts the resulting text (called translation unit) into binary code the CPU can execute.
Every **Variable** is **Defined** within some scope. A Variable cannot be referenced by name (a.k.a. **Symbol**) from outside of that scope.

Lexical scopes are defined with curly braces `{ }`. The scope of Function Arguments is the complete body of that function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called "Global" Vars.
# Comparison and Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
</tr>
</tbody>
</table>

+  plus  
-  minus 
*  mult  
/  divide 
%  modulo

Beware division:
- $5 / 10 \rightarrow 0$ whereas $5 / 10.0 \rightarrow 0.5$
- Division by 0 will cause a FPE

Don’t confuse & and &&..  
1 & 2 \rightarrow 0 whereas 1 && 2 \rightarrow <true>

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.
Assignment Operators

\[ x = y \quad \text{assign } y \text{ to } x \]
\[ x++ \quad \text{post-increment } x \]
\[ ++x \quad \text{pre-increment } x \]
\[ x-- \quad \text{post-decrement } x \]
\[ --x \quad \text{pre-decrement } x \]

Note the difference between \( ++x \) and \( x++ \) (high vs low priority (precedence)):

\[
\begin{align*}
\text{int } x &= 5; \\
\text{int } y; \\
y &= ++x; \\
\text{/* } x \text{ == 6, } y \text{ == 6 */}
\end{align*}
\]

\[
\begin{align*}
\text{int } x &= 5; \\
\text{int } y; \\
y &= x++; \\
\text{/* } x \text{ == 6, } y \text{ == 5 */}
\end{align*}
\]

Don’t confuse “=” and “==“:

\[
\begin{align*}
\text{int } x &= 5; \\
\text{if } (x == 6) \quad \text{/* } \text{false */} \\
\{ \\
\quad \text{/* } \ldots \text{ */}
\} \\
\text{/* } x \text{ is still 5 */}
\end{align*}
\]

\[
\begin{align*}
\text{int } x &= 5; \\
\text{if } (x = 6) \quad \text{/* always true */} \\
\{ \\
\quad \text{/* } x \text{ is now 6 */}
\} \\
\text{/* } \ldots \text{ */}
\end{align*}
\]
C Memory Pointers

- To discuss memory pointers, we need to talk first about the concept of memory

- We’ll conclude by touching on a couple of other C elements:
  - arrays, typedef, and structs
The “memory”

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its address. A one byte value can be stored in each slot.

Some data values span more than one slot, like the character string “Hello\n”

A type provides a logical meaning to a span of memory. Some simple types are:

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

char
char [10]
int
float
int64_t

a single character (1 slot)
an array of 10 characters
signed 4 byte integer
4 byte floating point
signed 8 byte integer
What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Declare a variable by giving it a name and specifying its type and optionally an initial value.

Variable x declared but undefined

Initial value

Name

Type is single character (char)

extern? static? const?

The compiler puts x and y somewhere in memory.

char x;
char y='e';

Symbol	Addr	Value
0	        
1	        
2	        
3	        
4	 x	 Some garbage
5	 y	 'e' (101)
6	        
7	        
8	        
9	        
10	       
11	       
12	       

What names are legal?
Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Architecture uses little-endian convention, since it stores the most significant byte first

An int requires 4 bytes

padding
One Quick Thing...

- I need one or two *volunteers* who can help my lab with expertise in MPI-enabled parallel computing.

- Our needs in Computational Dynamics:
  - Some loose ends need to be taken care of (code not entirely finished)
  - Code is very slow (simulations run two weeks)

- To view where we use parallel computing in the lab, look here:
  - [http://sbel.wisc.edu/Animations/](http://sbel.wisc.edu/Animations/)

- What you get in return: conference/journal papers

- If interested, please email me your CV
Quick Overview of C Programming
September 6, 2013

“There is no reason for any individual to have a computer in their home.”
Before We Get Started...

- Last time
  - Course logistics & syllabus overview

- Today
  - Cover in one lecture what normally covered in two weeks in a regular C course
  - Quick overview of C Programming
    - Essential reading: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
    - Read online primer
  - Acknowledgement: Slides on this C Intro include material from D. Zhang & L. Girod

- Other issues:
  - All in the waiting list have been cleared to register
  - You will get forum credentials by Monday end of day if not sooner
  - Learn@UW and class website should be up and running
  - Audio of 2013 will be available for each lecture
  - Video of 2012 to become available during the next week
The “memory”

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its Address. One byte Value can be stored in each slot.

Some data values span more than one slot, like the character string “Hello\n”

A Type provides a logical meaning to a span of memory. Some simple types are:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- **char**: a single character (1 slot)
- **char [10]**: an array of 10 characters
- **int**: signed 4 byte integer
- **float**: 4 byte floating point
- **int64_t**: signed 8 byte integer
What is a Variable?

A **Variable** names a place in memory where you store a **Value** of a certain **Type**.

You first **Declare** a variable by giving it a name and specifying its type and optionally an initial value. There is a subtle difference between “declaring” and “defining” a var.

```c
char x;
char y='e';
```

**Name**

What names are legal?

**Type is single character (char)**

extern? static? const?

**Initial value**

Variable x defined but uninitialized

The compiler puts x and y somewhere in memory.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char).

```c
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex.

An int requires 4 bytes.

In this picture, the architecture uses little-endian convention, since the least significant byte is stored at the lower address.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Memory, a more detailed view...

- A sequential list of **words**, starting from 0.
- Most often, but not always, a word is 4 bytes.
- Local variables are stored on the stack.
- Dynamically allocated memory is set aside on the heap (more on this later…)
- For multiple-byte variables, the address is that of the least significant byte (little endian).
Example...

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

900 904 908 912 916 920 924 928 932 936 940 944
Another Example

```cpp
#include <iostream>

int main() {
    char c[10];
    int d[10];
    int* darr;

    darr = (int *)(malloc(10*sizeof(int)));
    size_t sizeC = sizeof(c);
    size_t sizeD = sizeof(d);
    size_t sizeDarr = sizeof(darr);

    free(darr);
    return 0;
}
```

What is the value of:
- sizeC
- sizeD
- sizeDarr
Assume 32 bit OS.

**NOTE**: `sizeof` is a compile-time operator that returns the size, in multiples of the size of `char`, of the variable or parenthesized type-specifier that it precedes.
Can a C function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument? Are these equivalent?

```
float p = 2.0; /* p is 2.0 here */
p = pow(p, 5); /* p is 32.0 here */
```

```
float p = 2.0; /* p is 2.0 here */
pow_assign(p, 5); /* Is p is 32.0 here? */
```

Would this work?

Native function, to use you need
#include <math.h>

```
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; i < exp ; i++) {
        result = result * x;
    }
    x = result;
}
```
In C you can’t change the value of any variable passed as an argument in a function call...

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; i<exp; i++) {
        result = result * x;
    }
    x = result;
}
```

// a code snippet that uses above function
{
    float p=2.0;
    pow_assign(p, 5);
    // the value of p is 2 here...
}

In C, all arguments are passed by value

Keep in mind: pass by value requires the variable to be copied. That copy is then passed to the function. Sometime generating a copy can be expensive...

But, what if the argument is the address of a variable?
C Pointers

- What is a pointer?
  - A variable that contains the memory address of another variable or of a function

- In general, it is safe to assume that on 32 bit architectures pointers occupy one word
  - Pointers to int, char, float, void, etc. ("int*", "char*", "*float", "void"). They all occupy 4 bytes (one word).

- Pointers: *very* many bugs in C programs are traced back to mishandling of pointers...
Pointers (cont.)

- The need for pointers
  - Modifying a variable (its value) inside a function
    - The pointer to that variable is passed as an argument to the function
  - Passing large objects to functions without the overhead of copying them first
  - Accessing memory allocated on the heap
  - Passing functions as a function argument
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior:

- Very often the code will crash with a SEGV, that is, Segment Violation, or Segmentation Fault.

There are two general causes for these errors:

- Coding errors that end up setting the pointer to a strange number
- Use of a pointer that was at one time valid, but later became invalid

**Good practice:**

- Initialize pointers to 0 (or NULL). NULL is never a valid pointer value, but it is known to be invalid and means “no pointer set”.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 't'; /* valid? */
}
```

Will `ptr` be valid or invalid?
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

int main()
{
    char * ptr = get_pointer();
    *ptr = 't'; /* valid? */
    return 0;
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!

Here is what I get in VisualStudio when compiling:
main.cpp(3) : warning C4172: returning address of local variable or temporary
Example: What gets printed out?

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    printf("%p, %p, %p, %p, %p\n", &d, &c, &s, &p, arr);
    return 0;
}
```

• NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)

Q: What does get printed out by the `printf` call in the code snippet above?
Use of pointers, another example...

- Pass pointer parameters into function

```c
void swap(int *px, int *py) {
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int a = 5;
int b = 6;
swap(&a, &b);
```

- What will happen here?

```c
int * a;
int * b;
swap(a, b);
```

```bash
>> simple.cpp(17) : warning C4700: uninitialized local variable 'b' used
>> simple.cpp(17) : warning C4700: uninitialized local variable 'a' used
```
Dynamic Memory Allocation (Allocation on the Heap)

- Allows the program to determine how much memory it needs at run time and to allocate exactly the right amount of storage.
  - It is your responsibility to clean up after you (free the dynamic memory you allocated)

- The region of memory where dynamic allocation and deallocation of memory can take place is called the heap.
Dynamic Memory Allocation (cont.)

- Functions that come into play in conjunction with dynamic memory allocation

- An example of dynamic memory allocation

```c
void *malloc(size_t number_of_bytes);
    // allocates dynamic memory

size_t sizeof(type);
    // returns the number of bytes of type

void free(void * p);
    // releases dynamic memory allocated

int * ids;    // id arrays
int num_of_ids = 40;
ids = (int*) malloc( sizeof(int) * num_of_ids);
// ... do your work here...
free(ids);
```
Exercise

```cpp
#include<iostream>

int main()
{
    double *vals; // it’ll hold some values later on...
    int num_of_vals = 40;
    // vals = (double*) malloc( sizeof(*vals) * num_of_vals);
    vals = (double*) malloc( sizeof(double) * num_of_vals);

    int dummy = sizeof(vals);
    int dummier = sizeof(*vals);

    free(vals);
    return 0;
}
```

- How many bytes were allocated on the heap by the `malloc` operation?
- What is the value of `dummy`?
- What is the value of `dummier`?
- Would you get a compile-time error if you replaced the `malloc` operation by the one that is currently commented out?
More on Dynamic Memory Allocation

Recall that variables are allocated **statically** by having declared with a given size. This allocates them in the stack.

Allocating memory at run-time requires **dynamic** allocation. This allocates them on the heap.

```c
int * alloc_ints(size_t requested_count) {
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }

    /* big_array[0] through big_array[requested_count-1] are valid and zeroed. */
    return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can’t alloc

It’s OK to return this pointer. It will remain valid until it is freed with free(). However, it’s a bad practice to return it (if you need is somewhere else, declare and define it there…)

Caveats with Dynamic Memory

Dynamic memory is useful but be careful when you use it:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and `free()`-ed when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory causes a "memory leak".

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that was freed. Whenever you free memory you must be certain that you will not try to access it again.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
Moving on to other topics… What comes next:

- Creating logical layouts of different types (structs)
- Creating new types using typedef
- Using arrays
- Parsing C type names
Data Structures

- A data structure is a collection of one or more variables, possibly of different types.

- An example of student record

```c
struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};
```
Data Structures (cont.)

- A data structure is also a data type

```c
struct StudRecord my_record;
struct StudRecord * myPointer;
myPointer = & my_record;
```

- Accessing a field inside a data structure

```c
my_record.id = 10;
    \ or
myPointer->id = 10;
```
Allocating a data structure instance

This is a new type now

```c
struct StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(struct StudRecord));
pStudentRecord ->id = 10;
```

**IMPORTANT:** Never calculate the size of data structure yourself. Rely on the `sizeof()` function.
```cpp
#include <iostream>

int main() {
    struct StudRecord {
        char name[50];
        int id;
        int age;
        int major;
    };

    StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(StudRecord));

    strcpy(pStudentRecord->name, "Joe Doe");
pStudentRecord->id = 903107;
pStudentRecord->age = 20;
pStudentRecord->major = 643;

    return 0;
}
```
Example:
Use of malloc, memmove, struct

```cpp
#include <iostream>

int main()
{
    struct my_struct {
        int counter;
        float average;
        int in_use;
    } init;

    init.counter = 2;
    init.in_use = 0;
    init.average = 1.0f;

    // allocate heap memory for another my_struct variable
    my_struct* s;
    s = (my_struct*)malloc(sizeof(my_struct));
    memmove(s, &init, sizeof(init));

    return 0;
}
```
The “typedef” concept

```c
struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};

typedef struct StudRecord RECORD;

int main()
{
    RECORD my_record;
    strcpy_s(my_record.name, "Jean Doe");
    my_record.age = 21;
    my_record.id = 6114;

    RECORD* p = &my_record;
    p->major = 643;
    return 0;
}
```

Using typedef to improve readability…
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* access element 0, change its value */
x[0] = 'T';

/* pointer arithmetic to get 4th entry */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element; */
/* first element, or &(x[0]) */

/* 0-indexed for loop idiom */
define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}

/* x[3] == *(x+3) == ‘t’ (notice, it’s not ‘s’!) */

Q: What’s the difference between “char x[5]” and a declaration like “char *x”?

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..9)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char x [0]</td>
<td>100</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [1]</td>
<td>101</td>
<td>‘e’</td>
</tr>
<tr>
<td>char x [2]</td>
<td>102</td>
<td>‘s’</td>
</tr>
<tr>
<td>char x [3]</td>
<td>103</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [4]</td>
<td>104</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>
How to Parse and Define C Types

At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

- int x; /* int; */ typedef int T;
- int *x; /* pointer to int; */ typedef int *T;
- int x[10]; /* array of ints; */ typedef int T[10];
- int *x[10]; /* array of pointers to int; */ typedef int *T[10];
- int (*x)[10]; /* pointer to array of ints; */ typedef int (*T)[10];

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.
What are the values in x right before the return statement?
Rules of Precedence

- How do evaluate $x + 3*y[2]$?
  - According to the rules of precedence listed below, from highest

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>$x, y, f(x), a[x], x++, x--, \text{new typeof checked, unchecked}$</td>
</tr>
<tr>
<td>Unary</td>
<td>$+,-,!,\sim,++x,--x,(T)x$</td>
</tr>
<tr>
<td>Multiplicative</td>
<td>$*,/,%$</td>
</tr>
<tr>
<td>Additive</td>
<td>$+,-$</td>
</tr>
<tr>
<td>Shift</td>
<td>$&lt;&lt;,&gt;&gt;$</td>
</tr>
<tr>
<td>Relational and type testing</td>
<td>$&lt;,&gt;,&lt;=,&gt;=,\text{is, as}$</td>
</tr>
<tr>
<td>Equality</td>
<td>$==,!=,$</td>
</tr>
<tr>
<td>Logical</td>
<td>AND, &amp;</td>
</tr>
<tr>
<td>Logical</td>
<td>XOR, ^</td>
</tr>
<tr>
<td>Logical</td>
<td>OR,</td>
</tr>
<tr>
<td>Conditional</td>
<td>AND, &amp;&amp;</td>
</tr>
<tr>
<td>Conditional</td>
<td>OR,</td>
</tr>
<tr>
<td>Conditional</td>
<td>?::</td>
</tr>
<tr>
<td>Assignment</td>
<td>$=, *=, /=, %=, +=, -=, &lt;&lt;=, &gt;&gt;=, &amp;=, ^=,</td>
</tr>
</tbody>
</table>
Another less obvious construct is the “pointer to function” type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type)(const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

- The last argument is a comparison function.
- `const` means the function is not allowed to modify memory via this pointer.
- `void *` is a pointer to memory of unknown type.
- `size_t` is an unsigned int.
Why is Software Development Hard?

- **Complexity**: Every conditional ("if") doubles the number of paths through your code, every bit of state doubles possible states
  - Recommendation: reuse code with functions, avoid duplicate state variables

- **Mutability**: Software is easy to change. Great for rapid fixes… And rapid breakage…
  - Recommendation: tidy, readable code, easy to understand by inspection, provide *plenty* of meaningful comments.

- **Flexibility**: Problems can be solved in many different ways. Few hard constraints, easy to let your horses run wild
  - Recommendation: discipline and use of design patterns
Design Patterns

- A really good book if you are serious about this…
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];

    p = &d;
    *p = 10;
    c = (char)1;
    p = arr;
    *(p+1) = 5;
    p[0] = d;
    *( (char*)p + 1 ) = c;
    return 0;
}

Q: What are the values stored in arr? [assume little endian architecture]
Quiz [Cntd.]

```c
p = &d;
*p = 10;
c = (char)1;
p = arr;
*(p+1) = 5; // int* p;
p[0] = d;
*( (char*)p + 1 ) = c;
```

```
Question: arr[0] = ?
```

```plaintext
arr[0] = 10
arr[1] = 5
p = 904
d = 10
```
p = (int*) malloc(sizeof(int)*3);
s = (short)( *(p+2) );
free( p );
p=NULL;

Assumption: we have the same memory layout like in the previous example


Quiz [Cntd.]

```c
p = (int*) malloc(sizeof(int)*3);
short s = (short)( *(p+2) );
free( p );
```

Q: what will be the value of “s”, and why?

Q: what if you say p[2]=0 after you free the memory?
A: run time error.

Q: what if you do not call free(p)?
A: memory leak.
int dummy = 12399401
p = (int*) malloc(sizeof(int)*3);
p[2] = dummy * 3;
s = (short)( *(p+2) );
free( p );

Q: what is the value of “s” now, and why?

A:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>+2</td>
<td>+1</td>
<td>+0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[0] = 266</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[1] = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p = 932</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s = -26245</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d = 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p[2] = 37198203</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

p,3 932 int *
[0] -842150451 int
[1] -842150451 int
[2] 37198203 int
s -26245 short
End: Overview of C

Beginning: Discussion of Hardware Trends
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Conclusion, Quick Overview of C Programming

*gdb*

Logging into Euler

The **Eclipse** IDE

The Hardware/Software Interplay

September 9, 2013

“Be who you are and say what you feel, because those who mind don’t matter and those who matter don’t mind.”

Dr. Seuss
Before We Get Started…

- Last time
  - Brief overview of C
  - Most important concepts covered: pointers and memory layout

- Today
  - Wrap up quick overview of C Programming
  - Super quick intro to gdb (debugging tool under Linux)
  - Learn how to login and use Euler, the CPU/GPU cluster
  - Basic tidbits about how computers are organized and how they work

- First assignment made available on the class website later today
  - HW01 due on Mo, September 17, at 11:59 PM (Learn@UW cutoff time)
  - Post related questions to the forum
Another less obvious construct is the "pointer to function" type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));

/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type)(const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

- `void *` is a pointer to memory of unknown type.
- `size_t` is an unsigned int
- `const` means the function is not allowed to modify memory via this pointer.
- The last argument is a comparison function.
Why is Software Development Hard?

- **Complexity**: Every conditional (“if”) doubles the number of paths through your code, every bit of state doubles possible states
  - Recommendation: reuse code with functions, avoid duplicate state variables

- **Mutability**: Software is easy to change. Great for rapid fixes… And rapid breakage…
  - Recommendation: tidy, readable code, easy to understand by inspection, provide *plenty* of meaningful comments.

- **Flexibility**: Problems can be solved in many different ways. Few hard constraints, easy to let your horses run wild
  - Recommendation: discipline and use of design patterns
Design Patterns

- A good book if you are serious about programming
Quiz:
Usage of Pointers & Pointer Arithmetic

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    p = &d;
    *p = 10;
    c = (char)1;
    p = arr;
    *(p+1) = 5;
    p[0] = d;
    *( (char*)p + 1 ) = c;
    return 0;
}
```

Q: What are the values stored in arr? [assume little endian architecture]
Quiz [Cntd.]

p = &d;
*p = 10;
c = (char)1;

p = arr;
*(p+1) = 5;  // int* p;
p[0] = d;

*( (char*)p + 1 ) = c;

**Question:** arr[0] = ?
p = (int*) malloc(sizeof(int)*3);  
s = (short)( *(p+2) );  
free( p );  
p=NULL;

Assumption: we have the same memory layout like in the previous example
Quiz [Cntd.]

```
p = (int*) malloc(sizeof(int)*3);
short s = (short)( *(p+2) );
free( p );
P = NULL;
```

Q: what will be the value of “s”, and why?

Q: what if you say p[2]=0 after you free the memory?

A: code runs, weird stuff might happen

Q: what if you say p[2]=0 right after you set p to NULL?

A: code crashes (no compile time warning)

Q: what if you do not call free(p)?

A: memory leak.
int dummy = 12399401
p = (int*) malloc(sizeof(int)*3);
p[2] = dummy * 3;
s = (short)( *(p+2) );
free( p );

Q: what is the value of “s” now, and why?

A:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p,3</td>
<td>932</td>
<td>int</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>[0]</td>
<td>-842150451</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td>-842150451</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>37198203</td>
<td>int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>-26245</td>
<td>short</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

arr[0] = 266
arr[1] = 5
p = 932
s = -26245
c = 1
d = 10

p[2] = 37198203
Debugging on Euler
[with gdb]
gdb: Intro

- gdb: a utility that helps you debug your program
- Learning gdb is a great investment, boosts productivity
- A debugger will make a good programmer a better programmer
- In ME759, you should go beyond sprinkling "printf" here and there to try to debug your code
  - Avoid: Compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link,
Compiling a Program for gdb

- You need to compile with the “-g” option to be able to debug a program with gdb.

- The “-g” option adds debugging information to your program.
  
  ```bash
gcc -g -o hello hello.c
  ```
Running a Program with *gdb*

- To run a program called `progName` with *gdb* type
  
  ```
  >> gdb progName
  (gdb)
  ```

- Then set a breakpoint in the main function
  
  ```
  (gdb) break main
  ```

- A breakpoint is a marker in your program that will make the program stop and return control back to *gdb*

- Now run your program
  
  ```
  (gdb) run
  ```

- If your program has arguments, you can pass them after run.
Stepping Through your Program

- Your program will start running and when it reaches “main()” it will stop:
  (gdb)

- You can use the following commands to run your program step by step:
  (gdb) step
    It will run the next line of code and stop. If it is a function call, it will enter
    into it
  (gdb) next
    It will run the next line of code and stop. If it is a function call, it will not enter
    the function and it will go through it.

Example:
  (gdb) step
  (gdb) next
Printing the Value of a Variable

- The command:

  ```
  (gdb) print varName
  ... prints the value of a variable
  ```

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0} 
(gdb)
```
Setting Breakpoints

- A breakpoint is a location in a program where the execution stops in \texttt{gdb} and control is passed back to you.

- You can set breakpoints in a program in several ways:

  \begin{itemize}
  \item \texttt{(gdb) break functionName}
    \begin{itemize}
    \item Set a breakpoint in a function. E.g.
    \begin{itemize}
    \item \texttt{(gdb) break main}
    \end{itemize}
    \end{itemize}
  \item \texttt{(gdb) break lineNumber}
    \begin{itemize}
    \item Set a break point at a line in the current file. E.g.
    \begin{itemize}
    \item \texttt{(gdb) break 66}
    \end{itemize}
    \end{itemize}
    \begin{itemize}
    \item It will set a break point in line 66 of the current file.
    \end{itemize}
  \item \texttt{(gdb) break fileName:lineNumber}
    \begin{itemize}
    \item It will set a break point at a line in a specific file. E.g.
    \begin{itemize}
    \item \texttt{(gdb) break hello.c:78}
    \end{itemize}
    \end{itemize}
  \item \texttt{(gdb) break fileName:functionName}
    \begin{itemize}
    \item It will set a break point in a function in a specific file. E.g.
    \begin{itemize}
    \item \texttt{(gdb) break subdivision.c:partialSum}
    \end{itemize}
    \end{itemize}
  \end{itemize}
Watching a Variable

- Many times you want to keep an eye on a variable that for some reason assumes a value that is not in line with expectations.

- To that end, you can “watch” a variable and have the code break as soon as the variable is read or changed.

- You can set breakpoints in a program in several ways:

  (gdb) watch varName
  Program breaks whenever varName gets written by the program

  (gdb) rwatch varName
  Program breaks whenever varName gets read by the program

  (gdb) awatch varName
  Program breaks whenever varName gets read/written by the program

  (gdb) info watchpoints
  You get a list of all watchpoints, breakpoints, and catchpoints in your program.
Example:
[watching a variable]

```cpp
#include <iostream>

int main()
{
    int arr[2]={266, 5};

    int * p;
    short  s;

    p = (int*) malloc(sizeof(int)*3);


    s = (short)( *(p+2) );

    free( p );

    p=NULL;

    p[0] = 5;
    return 0;
}
```
Example: Watching Variable “s”

- Below is a copy-and-paste from gdb, for our short program

```plaintext
(gdb) awatch s
Hardware access (read/write) watchpoint 2: s
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 2: s
Old value = 0
New value = 15
main () at pointerArithm.cpp:15
15       free( p );
```
Regaining the Control

- When you type
  \(\text{(gdb) run}\)
  the program will start running and it will stop at a breakpoint

- If the program is running without stopping, you can regain control again typing \text{ctrl-c}\n
- When you type
  \(\text{(gdb) continue}\)
  the program will run until it hits the next breakpoint, or exits
Where Are You?

- The command
  
  \[(gdb) \text{where}\]

  Will print the current function being executed and the chain of functions that are calling that function.

  This is also called the backtrace.

Example:

\[(gdb) \text{where}\]

#0 main () at test_mysting.c:22

\[(gdb)\]
Seeing Code Around You...

- The command `list` shows you code around the location where the execution is “break-ed”
  ```
  (gdb) list
  It will print, by default, 10 lines of code.
  ```

There are several flavors:

- `(gdb) list lineNumber`
  ...prints code around a certain line number

- `(gdb) list functionName`
  ...prints lines of code around the beginning of a function

- `(gdb) set listsize someNumber`
  ...controls the number of lines showed with `list` command
Exiting `gdb`

- The command “quit” exits `gdb`.

  ```
  (gdb) quit
  The program is running. Exit anyway?
  (y or n) y
  ```
Debugging a Crashed Program

- Also called “postmortem debugging”

- When a program segfaults, it writes a core file.
  
  ```bash
  bash-4.1$ ./hello
  Segmentation Fault (core dumped)
  bash-4.1$
  ```

- The core is a file that contains a snapshot of the state of the program at the time of the crash
  - Information includes what function the program was running upon crash
Example: [Code crashing]

```cpp
#include <iostream>

int main(){
    int arr[2]={266,5};

    int * p;
    short s;

    p = (int*) malloc(sizeof(int)*3);


    s = (short)( *(p+2) ) ;

    free( p );

    p=NULL;

    p[0] = 5;
    return 0;
}
```

This is why it’s crashing…
Running gdb on a Segmentation fault

- Here’s what gdb says when running the code...

```
[negrut@euler CodeBits]$ gdb badPointerArithm.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
(gdb) run
Starting program: /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out...done.
(gdb) run
Starting program: /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out
warning: the debug information found in "/usr/lib/debug//lib64/libc-2.12.so.debug" does not match "/lib64/libc.so.6" (CRC mismatch).
warning: the debug information found in "/usr/lib/debug/lib64/libc-2.12.so.debug" does not match "/lib64/libc.so.6" (CRC mismatch).

Program received signal SIGSEGV, Segmentation fault.
0x0000000000000400641 in main () at pointerArithm.cpp:19
19    p[0] = 5;
(gdb)
```
Debugging – Departing Thoughts

- Debug like a pro (don’t use `printf`...)

- `dbg` can save you tons of time

- If you want to have a GUI slapped on top of `gdb`, use “`ddd`” on Euler

- Under Windows, VisualStudio has an excellent debugger
“I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.“

The editor in charge of business books for Prentice Hall, 1957.
Before We Get Started…

- Last time
  - Wrap up quick overview of C Programming
  - Super quick intro to gdb (debugging tool under Linux)
  - Learn how to login and use Euler, the CPU/GPU cluster

- Today
  - Basic tidbits about how computers are organized and how they work

- Reading Assignment
  - Read first 27 pages of the primer available on the website
Basic Elements Related to the Hardware/Software Interplay
Elements of Processor Architecture
Today’s Computer

- Follows paradigm formalized by Von Neumann in late 1940s

- The von Neumann model:
  - There is no distinction between data and instructions
  - Data and instructions are stored in memory as a string of 0 and 1 bits
    - Instructions are fetched + decoded + executed
    - Data is used to produce results according to rules specified by the instructions
From Code to Instructions

- There is a difference between a line of code and a processor instruction.
- Example:
  - Line of C code:
    ```
a[4] = delta + a[3]; //line of C code
    ```
  - MIPS assembly code generated by the compiler:
    ```
lw $t0, 12($s2)  // reg $t0 gets value stored 12 bytes from address in $s2
add$t0, $s4, $t0  // reg $t0 gets the sum of values stored in $s4 and $t0
sw $t0, 16($s2)  // a[4] gets the sum delta + a[3]
```
- Set of three corresponding MIPS instructions produced by the compiler:

```
100011100100100000000000000001100
000000101000100010000100000000010000
1010111001001000000000000000010000
```
From Code to Instructions

- **C code** – what you write to implement an algorithm
- **Assembly code** – what your code gets translated into by the compiler
- **Instructions** – what the assembly code gets translated into by the compiler

**Observations:**
- The compiler typically goes from C code directly to machine instructions
- Machine instructions: what you see in an editor like **notepad** or **vim** or **emacs** if you open up an executable file
- There is a one-to-one correspondence between an assembly line of code and an instruction (most of the time)
- Assembly line of code can be regarded as an instruction that is expressed in a way that humans can relatively easy figure out what happens
- Back in the day people wrote assembly code
- Today coding in assembly done only for the super critical parts of a program if you want to optimize and don’t trust the compiler
Instruction Set Architecture (ISA)

- The same line a C code can lead to a different set of instructions on two different computers

- This is so because two CPUs might draw on two different Instruction Set Architectures (ISA)

- ISA: defines the “language” that expresses at a very low level the actions of a processor

- Example:
  - Microsoft’s Surface Tablet
    - RT version: uses a Tegra chip, which implements an ARM Instruction Set
    - Pro version: uses an Intel Atom chip, which implements x86 Instruction Set
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

C code

```c
int main()
{
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

x86 ISA

call __main
fldl LC0
fstpl -40(%ebp)
fldl LC1
fstpl -32(%ebp)
fldl LC2
fstpl -24(%ebp)
faddl -24(%ebp)
fldl LC0
fmulp %st, %st(1)
stpl -16(%ebp)
movl $0, %eax
addl $36, %esp
popl %ecx
popl %ebp
leal -4(%ecx), %esp
ret

LC0:
.long 387883269
.long 1066524452
.align 8

LC1:
.long 0
.long 1078853632
.align 8

LC2:
.long 0
.long 1079902208

MIPS ISA

```
main:
.frame $fp,48,$31 # vars= 32, regs= 1/0, args= 0, gp= 8
.mask 0x40000000,-4
.fmask0x00000000,0
.set noreorder
.set nomacro
addiu $sp,$sp,-48
sw $fp,44($sp)
move $fp,$sp
lui $2,%hi($LC0)
lwc1

... mul.d $f0,$f2,$f0
swc1 $f0,32($fp)
swc1 $f1,36($fp)
move $2,$0
move $sp,$fp
lw $fp,44($sp)
addiu $sp,$sp,48
j $31

$LC0:
.word 3649767765
.word 1066523892
.align 3

$LC1:
.word 0
.word 1078853632
.align 3

$LC2:
.word 0
.word 1079902208
.ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
```
Instruction Set Architecture vs. Chip Microarchitecture

- ISA – can be regarded as a standard
  - Specifies what a processor should be able to do
    - Load, store, jump on less than, etc.

- Microarchitecture – how the silicon is organized to implement the functionality promised by ISA

- Example:
  - Intel and AMD both use the x86 ISA
  - Nonetheless, they have different microarchitectures
RISC vs. CISC

- **RISC Architecture – Reduced Instruction Set Computing Architecture**
  - Usually each instruction is coded into a set of 32 bits
  - Recently a move to 64 bits
  - Each executable has fixed length instruction be it 32 or 64 (no mixing)
  - The key attribute: the length of the instruction is fixed
  - Promoted by: ARM Holding, company that started as ARM (Advanced RISC Machines)
    - Use in: embedded systems, smart phones – Intel, NVIDIA, Samsung, Qualcomm, Texas Instruments
    - Somewhere between 8 and 10 billion chips based on ARM manufactured annually

- **CISC Architecture – Complex Instruction Set Computing Architecture**
  - Instructions have various lengths
    - Examples: 32 bit instruction followed by 256 bit instruction followed later on by 128 bit instruction, etc.
  - Intel’s X86 is the most common example
  - Promoted by: Intel, AMD
    - Used in: laptops, desktops, workstations, supercomputers
RISC vs. CISC

- RISC is simpler to comprehend, provision for, and work with
- Decoding CISC leads to extra power consumption and makes things more complicated
- A CISC instruction is usually broken down into several micro-operations (uops)
- CISC Architectures invite spaghetti type evolution of the ISA and require complex microarchitecture
  - Provide the freedom to do as you wish
The CPU’s Control Unit (CU)

- Think of a CPU as a big kitchen
  - A work order comes in (this is an instruction)
  - The cook (this is the ALU) starts to cook a meal
  - Some ingredients are needed: meat, spinach, potatoes, etc. (this is the data)
  - Some ready to eat product goes out the kitchen: a soup (this is the result)

- The cook, the passing of meat, passing of pasta, the movement of the sautéed meat to chopping board, boiling of pasta, etc. – they happen in a coordinated fashion (based on a kitchen clock) and is managed by the CU

- The CU manages/coordinates/controls based on information in the work order (the instruction)
The FDX Cycle

- FDX stands for Fetch-Decode-Execute
- This is what the CPU keeps doing to execute a sequence of instructions that combine to make up a program

- Fetch: an instruction is fetched from memory
  - Recall that it will look like this (on 32 bits, MIPS, \texttt{lw \$t0, 12($s2) }):
    
    \begin{verbatim}
    10001110010010000000000000001100
    \end{verbatim}

- Decode: this strings of 1s and 0s are decoded by the CU
  - Example: here’s an “I” (eye) type instruction, made up of four fields
Decoding: Instructions Types

- Three types of instructions in MIPS ISA
  - Type I
  - Type R
  - Type J
Type I (MIPS ISA)

- The first six bits encode the basic operation; i.e., the opcode, that needs to be completed
  - Example adding two numbers (000000), subtracting two numbers (000001), dividing two numbers (000011), etc.
- The next group of five bits indicates in which register the first operand is stored
- The subsequent group of five bits indicates the register where the second operand is stored.
- Some instructions require an address or some constant offset. This information is stored in the last 16 bits
Type R (MIPS ISA)

- Type R has the same first three fields op, rs, rt like I-type

- Packs three additional fields:
  - Five bit rd field (register destination)
  - Five bit shamt field (shift amount)
  - Six bit funct field, which is a function code that further qualifies the opcode
FDX Cycle: The Execution Part
It All Boils Down to Transistors…

- Why are transistors important?

- Transistors can be organized to produce complex logical units that have the ability to execute instructions

- More transistors increase opportunities for building/implementing in silicon functional units that can operate at the same time towards a shared goal
Transistors at Work: AND, OR, NOT

- NOT logical operation is implemented using one transistor
- AND and OR logical ops require two transistors

 Truth tables for AND, OR, and NOT

<table>
<thead>
<tr>
<th>AND</th>
<th>$i_{n_2}=0$</th>
<th>$i_{n_2}=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_{n_1}=0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i_{n_1}=1$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>$i_{n_2}=0$</th>
<th>$i_{n_2}=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_{n_1}=0$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$i_{n_1}=1$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>$i_{n_1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_{n_1}=0$</td>
<td>1</td>
</tr>
<tr>
<td>$i_{n_1}=1$</td>
<td>0</td>
</tr>
</tbody>
</table>
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1

<table>
<thead>
<tr>
<th>$in_1$</th>
<th>$in_2$</th>
<th>$in_3$</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic Equation:

$$out = \overline{in_3} + in_2 \cdot in_1$$

- Solution: digital logic block is a combination of AND, OR, and NOT gates
  - The NOT is represented as a circle O applied to signals moving down the bus
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

### Truth Table

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁</td>
<td>in₂</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Logic Equation:

$$\text{CarryOut} = (\text{in}_1 \cdot \text{CarryIn}) + (\text{in}_2 \cdot \text{CarryIn}) + (\text{in}_1 \cdot \text{in}_2)$$
Integrated Circuits-A One Bit Combo: OR, AND, 1 Bit Adder

- 1 Bit Adder, the Sum part

- Combo: OR, AND, 1 Bit Sum
  - Controlled by the input “Operation”
Integrated Circuits: Ripple Design of 32 Bit Combo

- Combine 32 of the 1 bit combos in an array of logic elements
  - Get one 32 bit unit that can do OR, AND, +
Integrated Circuits: From Transistors to CPU

- **Transistor** → **Gate**
  - **Logical Unit**: Examples: AND, OR, NOT
  - Example: \( \text{out} = in_1 + in_2 \)

- **Mux (selector)**
  - Requires a "Control Signal" as input

- **Complex Combinational Block**
  - Example: One bit adder

- **Array of Logic Elements**
  - Example: 32 bit adder

From simple to complex...
Every 18 months, the number of transistors per unit area doubles (Moore’s Law)
  - Current technology (2013): feature length is 22 nm
  - Next wave (2014): feature length is 14 nm

Example
  - NVIDIA Fermi architecture of 2010:
    - 40 nm technology
    - Chips w/ 3 billion transistors → more than 500 scalar processors, 0.5 TFlops
  - NVIDIA Fermi architecture 2012:
    - 28 nm technology
    - Chips w/ 7 billion transistors → more than 2000 scalar processors, 1.5 TFlops

It All Boils Down to Transistors…
Registers
Registers

- Instruction cycle: fetch-decode-execute (FDX)
- CU – responsible for controlling the process that will deliver the request baked into the instruction
- ALU – does the busy work to fulfill the request put forward by the instruction
- The instruction that is being executed should be stored somewhere
- Fulfilling the requests baked into an instruction usually involves handling input values and generates output values
  - This data needs to be stored somewhere
Registers

- Registers, quick facts:
  - A register is an entity whose role is that of storing information
  - A register is the type of storage with shortest latency – it’s closest to the ALU
  - Typically, one cannot control what gets kept in registers (with a few exceptions)

- The number AND size of registers used are specific to a ISA
  - Prime example of how ISA decides on something and the microarchitecture has to do what it takes to implement this design decision

- In MPIS ISA: there are 32 registers of 32 bits that are used to store critical information
Register Types

- Discussion herein covers only several register types typically encountered in a CPU (abbreviation in parenthesis)
  - List not comprehensive, showing only the more important ones

- Instruction register (IR) – a register that holds the instruction that is executed
  - Sometimes known as “current instruction register” CIR

- Program Counter (PC) – a register that holds the address of the next instruction that will be executed
  - NOTE: unlike IR, PC contains an *address* of an instruction, not the actual instruction
Register Types [Cntd.]

- Memory Data Register (MDR) – register that holds data that has been read in from main memory or, alternatively, produced by the CPU and waiting to be stored in main memory.

- Memory Address Register (MAR) – the address of the memory location in main memory (RAM) where input/output data is supposed to be read in/written out.
  - NOTE: unlike MDR, MAR contains an *address* of a location in memory, not actual data.

- Return Address (RA) – the address where upon finishing a sequence of instructions, the execution should return and commence with the execution of subsequent instruction.
- Registers on previous two slides are a staple in most chip designs.

- There are several other registers that are common to many chip designs yet they are encountered in different numbers.

- Since they come in larger numbers they don’t have an acronym:
  - Registers for Subroutine Arguments (4) – a0 through a3
  - Registers for temporary variables (10) – t0 through t9
  - Registers for saved temporary variables (8) – s0 through s7
    - Saved between function calls.
Register Types [Cntd.]

- Several other registers are involved in handling function calls

- Summarized below, but their meaning is only apparent in conjunction with the organization of the virtual memory

  - Global Pointer (gp) – a register that holds an address that points to the middle of a block of memory in the static data segment

  - Stack Pointer (sp) – a register that holds an address that points to the last location on the stack (top of the stack)

  - Frame Pointer (fp) - a register that holds an address that points to the beginning of the procedure frame (for instance, the previous sp before this function changed it’s value)
Register, Departing Thoughts

- **Examples:**
  - In 32 bit MIPS ISA, there are 32 registers
  - On a GTX580 NVIDIA card there are more than 500,000 32 bit temporary variable registers to keep busy 512 Scalar Processors (SPs) that made up 16 Stream Multiprocessors (SMs)

- Registers are very precious resources

- Increasing their number is not straightforward
  - Need to change the design of the chip (the microarchitecture)
  - Need to work out the control flow
Reading Assignment

- Read the primer document to learn about
  - The FDX (fetch-decode-execute) cycle
  - The bus

- Read first 27 pages of the document
  - Post comments on the forum, there is a discussion thread dedicated to the primer

- URL:
  http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf
Before We Get Started…

- Last time
  - ISA
  - From code to instructions
  - Transistors, and why we need lots of them
  - Registers

- Today
  - Pipelining
  - SRAM/DRAM
  - Discussion of memory: caches and main memory

- Assignments
  - First assignment due on Monday
    - Try to upload a fake zipped file to see if you can do so at Learn@UW
  - Read first 27 pages of the primer available on the website. Post suggestions on forum
Pipelining
Pipelining, or the Assembly Line Concept

- Henry Ford: perfected the assembly line idea on an industrial scale and in the process shaped the automotive industry (Ford Model T)

- Vehicle assembly line: a good example of a pipelined process
  - The output of one stage (station) becomes the input for the downstream stage (station)
  - It is bad if one station takes too long to produce its output since all the other stations idle a bit at each cycle of the production
  - “cycle” is the time it takes from the moment a station gets its input to the moment the output is out of the station
  - In this setup, an instruction (vehicle) gets executed (assembled) during each cycle
FDX cycle: carried out in conjunction with each instruction
- Fetch, Decode, Execute

A closer look at what gets fetched (instructions and data) and then what happens upon execution leads to a generic five stage process associated with an instruction

“generic” means that in a first order approximation, these five stages can represent all instructions, although some instructions might not have all five stages:
- Stage 1: Fetch an instruction
- Stage 2: Decode the instruction while reading registers
- Stage 3: Execute the operation (Ex.: might be a request to calculate an address)
- Stage 4: Data access
- Stage 5: Write-back into register file

NOTE: In general, these are the five generic stages of a RISC architecture
- MIPS is a special case of RISC

[Patterson, 4th edition]→
Not all types of instructions require all five stages

Example, based on the MIPS ISA:

- **Number of cycles required**
  - Load a word (lw): five clock cycles. In absolute terms, 800 ps (the register read/write burn only half of the time in a cycle)
  - Store a word (sw) as well as R-format instructions: four cycles
    - sw: 700 ps
    - R-format instructions shown: 600 ps
  - J-type instruction (branch-on-equal): 3 cycles. In absolute terms, 500 ps

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load word (lw)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>800 ps</td>
</tr>
<tr>
<td>Store word (sw)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>700 ps</td>
</tr>
<tr>
<td>R-Format (add, sub)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>600 ps</td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>500 ps</td>
</tr>
</tbody>
</table>

[Paterson, 4th edition]
Pipelining, Basic Idea

- At the cornerstone of pipelining is the observation that the following tasks can be worked upon simultaneously when processing five instructions:
  - Instruction 1 is in the 5th stage of the FDX cycle
  - Instruction 2 is in the 4th stage of the FDX cycle
  - Instruction 3 is in the 3rd stage of the FDX cycle
  - Instruction 4 is in the 2nd stage of the FDX cycle
  - Instruction 5 is in the 1st stage of the FDX cycle

- The above is a five stage pipeline

- An ideal situation is when each of these stages takes the same amount of time for completion
  - The pipeline is balanced

- If there is a stage that takes a significantly longer time since it does significantly more than the other stages, it should be broken into two and the length of the pipeline increases by one stage
Example: Streaming for execution 3 SW instructions

\text{sw} \quad \text{\$t0, 0(\$s2)}

\text{sw} \quad \text{\$t1, 32(\$s2)}

\text{sw} \quad \text{\$t2, 64(\$s2)}

- Case 1: No pipelining – 2100 picoseconds [ps]
Example: Streaming for execution 3 SW instructions

\[
\begin{align*}
&\text{sw } \$t0, \ 0(\$s2) \\
&\text{sw } \$t1, \ 32(\$s2) \\
&\text{sw } \$t2, \ 64(\$s2)
\end{align*}
\]

- Case 2: With pipelining – 1200 picoseconds [ps]
Pipelining, Benefits

- Assume that you have
  1. A very large number of instructions
  2. Balanced stages
     - Not the case in our example, since “Reg” wasted half of the pipeline stage time
  3. A pipeline that is larger than or equal to the number “p” of stages associated with the typical ISA instruction

- If 1 through 3 above hold, in a first order approximation, the speed-up you get out of pipelining is approximately “p”

- Benefit stems from parallel processing of FDX stages
  - This kind of parallel processing of stages is transparent to the user
    - Unlike GPU or multicore parallel computing, you don’t have to do anything to benefit of it
Pipelining, Benefits

- Why the speedup?
  - Goes back to computing in parallel
  - All instructions have $p$ stages and you have a pipeline of length $p$
  - Nonpipelined execution of $N$ instructions: $N \times p$ cycles needed to finish
  - Pipelined execution of $N$ instruction: during each cycle, $p$ stages out of $N \times p$ are executed $\Rightarrow$ you only need $N$ cycles
  - This glosses over the fact that you need to prime the pipeline and there is a shutdown sequence that sees pipeline stages being empty
Pipelining, Good to Remember

- The amount of time required to complete one stage of the pipeline: one cycle
- Pipelined processor: one instruction processed in each cycle
- Nonpipelined processor: several cycles required to process an instruction:
  - Four cycles for SW, five for LW, four for add, etc.

- Important Remark:
  - Pipelining does not decrease the time to process one instruction but rather it increases the throughput of the processor
Pipelining Hazards

- Q: if deep pipelines are good, why not have them bigger and bigger?
- A: deep pipelines plagued by “pipelining hazard”

- These “hazards” come in three flavors
  - Structural hazards
  - Data hazards
  - Control hazards
Pipeline Structural Hazards [1/2]

- The instruction pipelining analogy with the vehicle assembly line breaks down at the following point:
  - A real world assembly line assembles the same product for a period of time
  - Might be quickly reconfigured to assemble a different product
  - Instruction pipelining must process a broad spectrum of instructions that come one after another
    - Example: A J-type instruction coming after a R-type instruction, which comes after three I-Type instructions
    - If they were the same instructions (vehicles), designing a pipeline (assembly line) is straightforward

- A structural hazard refers to the possibility of having a combination of instructions in the pipeline that are contending for the same piece of hardware
  - Not encountered when you assemble the same car model (things are deterministic in this case)
**Pipeline Structural Hazards** [2/2]

- Possible Scenario: you have a six stage pipeline and the instruction in stage 1 and instruction in stage 5 both need to use the same register to store a temporary variable.
  - Resolution: there should be enough registers provisioned so that no combination of instructions in the pipeline leads to RAW, WAR, etc. type issue
  - Alternative solution: serialize the access, basically stall the pipeline for a cycle so that there is no contention

- Note:
  - Adding more registers is a static solution; expensive and very consequential (requires a chip design change)
  - Stalling the pipeline at run time is a dynamic solution that is inexpensive but slows down the execution
Pipeline Data Hazards [1/2]

- Consider the following example in a five stage pipeline setup:

```
add $t0, $t2, $t4  # $t0 = $t2 + $t4
addi $t3, $t0, 16  # $t3 = $t0 + 16 ("add immediate")
```

- The first instruction is processed in five stages
- Its output (value stored in register $t0) is needed in the very next instruction
- Data hazard: unavailability of $t0 to the second instruction, which references this register
- Resolution (less than ideal)
  - Pipeline stalls to wait for the first instruction to fully complete
Pipeline Data Hazards [2/2]

```
add $t0, $t2, $t4  # $t0 = $t2 + $t4
addi $t3, $t0, 16  # $t3 = $t0 + 16 ("add immediate")
```

- Alternative [the good] Resolution: use “forwarding” or “bypassing”

- Key observation: the value that will eventually be placed in $t0 is available after stage 3 of the pipeline (where the ALU actually computes this value)

- Provide the means for that value in the ALU to be made available to other stages of the pipeline right away
  - Nice thing: avoids stalling - don’t have to wait several other cycles before the value made its way in $t0
  - This process is called a forwarding of the value

- Supporting forwarding does not guarantee resolution of all scenarios
  - On relatively rare occasions the pipeline ends up stalled for a couple of cycles

- Note that the compiler can sometimes help by re-ordering instructions
  - Not always possible
Pipeline Control Hazards [Setup]

- What happens when there is an “if” statement in a piece of C code?

- A corresponding machine instruction decides the program flow
  - Specifically, should the “if” branch be taken or not?

- Processing this very instruction to figure out the next instruction (branch or no-branch) will take a number of cycles

- Should the pipeline stall while this instruction is fully processed and the branching decision becomes clear?
  - If yes: approach works, but it is slow
  - If no: you rely on branch prediction and proceed fast but cautiously
Pipeline Control Hazards: Branch Prediction

- Note that when you predict wrong you have to discard instruction[s] executed speculatively and take the correct execution path

- Static Branch Prediction (1\textsuperscript{st} strategy out of two):
  - Always predict that the branch will not be taken and schedule accordingly
  - There are other heuristics for proceeding: for instance, for a do-while construct it makes sense to always be jumping back at the beginning of the loop
    - Similar heuristics can be produced in other scenarios (a “for” loop, for instance)

- Dynamic Branch Prediction (2\textsuperscript{nd} strategy out of two):
  - At a branching point, the branch/no-branch decision can change during the life of a program based on recent history
  - In some cases branch prediction accuracy hits 90%
Pipelining vs. Multiple-Issue

- Pipelining should not be confused with “Multiple-Issue” as an alternative way of speeding up execution.
- A Multiple-Issue processor core is capable of processing more than one instruction at each cycle.
- Two examples to show when this might come in handy:
  - Example 1: performing an integer operation while performing a floating point operation – they require different resources and therefore can proceed simultaneously.
  - Example 2: the two lines of C code below lead to a set of instructions that can be executed at the same time.

```c
int a, b, c, d;
//some code setting a and b here
int c = a + b;
int d = a - b;
```
Pipelining vs. Multiple-Issue

- Multiple-Issue can be done statically or dynamically
  - Static multiple-issue:
    - Predefined, doesn’t change at run time
    - Who uses it: NVIDIA - very common in parallel computing on the GPU
  - Dynamic multiple-issue:
    - Changed at run time by taking account hardware resources that can take additional work
    - Who uses it: Intel, uses it heavily

- NOTE: Both pipelining and multiple-issue are presentations of what is called Instruction-Level Parallelism (ILP)
Attributes of Dynamic Multiple-Issue

- Instructions are issued from one instruction stream
- More than one instruction is processed by the same core in the same clock cycle
- The data dependencies between instruction being processed takes place at run time
- NOTE: sometimes called a superscalar architecture
Measuring Computing Performance
Nomenclature

- **Program Execution Time** – sometimes called *wall clock time*, elapsed time, response time
  - Most meaningful indicator of performance
  - Amount of time from the beginning of a program to the end of the program
  - Includes (factors in) all the housekeeping (running other programs, OS tasks, etc.) that the CPU has to do while running the said program

- **CPU Execution Time**
  - Like “Program Execution Time” but counting only the amount of time that is effectively dedicated to the said program
  - Requires a profiling tool to gauge

- On a dedicated machine; i.e., a quiet machine, Program Execution Time and CPU Execution Time would virtually be identical

[Patterson, 4th edition]
Nomenclature [Cntd.]

- Qualifying CPU Execution Time further:
  - User time – the time spent processing instructions compiled out of code generated by the user or in libraries that are directly called by user code
  - System time – time spent in support of the user’s program but in instructions that were not generated out of code written by the user
    - OS support: open file for writing/reading, throw an exception, etc.
  - The line between the user time and system time is somewhat blurred, hard to delineate these two times at times

- Clock cycle, clock, cycle, tick – the length of the period for the processor clock; typically a constant value dictated by the frequency at which the processor operates
  - Example: 2 GHz processor has clock cycle of 500 picoseconds
The CPU Performance Equation

The three ingredients of the CPU Performance Equation:

- Number of instructions that your program executes (Instruction Count)
- Average number of clock cycles per instructions (CPI)
- Clock Cycle Time

The CPU Performance Equation reads:

CPU Exec. Time = Instruction Count × CPI × Clock Cycle Time

Alternatively, using the clock rate

CPU Exec. Time = \( \frac{\text{Instruction Count} \times CPI}{\text{Clock Rate}} \)

[Patternson, 4th edition]
CPU Performance: How can we improve it?

- To improve performance the product of three factors should be reduced

- For a long time, we surfed the wave of “let’s increase the frequency”; i.e., reduce clock cycle time
  - We eventually hit a wall this way (the “Power Wall”)

- As repeatedly demonstrated in practice, reducing the Instruction Count (IC) often times leads to an increase in CPI. And the other way around.
  - Ongoing argument: whether RISC or CISC is the better ISA
    - The former is simple and therefore can be optimized easily. Yet it requires a large number of instructions to accomplish something in your C code
    - The latter is mind boggling complex but instructions are very expressive. Leads to few but expensive instructions to accomplish something in your C code
    - Specific example: ARM vs. x86
SPEC CPU Benchmarks

- There are benchmarks used to gauge the performance of a processor

- Idea: gather a collection of programs that use a good mix of instructions and flex the muscles of the chip

- These programs are meant to be representative of a class of applications that people are commonly using and not favor a chip manufacturer at the expense of another one

- Example: a compiler is a program that is used extensively, so it makes sense to have it included in the benchmark

- Two common benchmarks:
  - For programs that are dominated by floating point operations (CFP2006)
  - A second one is meant to be a representative sample of programs that are dominated by integer arithmetic (CINT2006)
### SPEC CPU Benchmark: Example, highlights AMD performance

<table>
<thead>
<tr>
<th>CINT2006 Programs</th>
<th>AMD Opteron X4 – 2356 (Barcelona)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td><strong>Instruction count [×10^9]</strong></td>
</tr>
<tr>
<td>Interpreted string processing</td>
<td>perl</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>bzip2</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>gcc</td>
</tr>
<tr>
<td>Combinational optimization</td>
<td>mcf</td>
</tr>
<tr>
<td>Go game (AI)</td>
<td>go</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>hmmer</td>
</tr>
<tr>
<td>Chess game (AI)</td>
<td>sjeng</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>libquantum</td>
</tr>
<tr>
<td>Video compression</td>
<td>h264avc</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>omnitpp</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>aster</td>
</tr>
<tr>
<td>XML parsing</td>
<td>xatancbmk</td>
</tr>
</tbody>
</table>

[Patterson, 4th edition]→
SPEC CPU Benchmark:
Example, highlights AMD performance

- Comments:
  - There are programs for which the CPI is less than 1.
    - Suggests that multiple issue is at play
  - Why are there programs with CPI of 10?
    - The pipeline stalls a lot, most likely due to repeated cache misses and system memory transactions

[Patterson, 4th edition]
The Most Important Lesson in ME759

- The cost of memory transactions trumps by far the number crunching cost

- Number crunching is free, sustaining the number crunching is the hard part
Memory Aspects
SRAM

- **SRAM – Static Random Access Memory**
  - Integrated circuit whose elements combine to make up memory arrays
  - “Element”: is a special circuit, called flip-flop
  - One flip-flop requires four to six transistors
  - Each of these elements stores on bit of information
  - Very short access time: $\approx 1$ ns (order of magnitude)
  - Uniform access time of any element in the array (yet it’s different to write than to read)
  - “Static” refers to the fact that once set, the element stores the value set as long as the element is powered
  - Bulky, since a storing element if “fat”; problematic to store a lot per unit area (compared to DRAM)
  - Expensive, since it requires four to six more transistors and different layout and support requirements
EXAMPLE: SRAM

- SRAM chip above stores 4 million elements, each with 8 bits of data ⇒ 4 MB
- To this end, you need
  - 22 bits to specify the address of the 8 bit slot that you want to address
  - Another control input that selects this chip (“Chip select” control signal)
  - A signal to indicate, when applicable, a write operation (Write enable)
  - A signal to indicate, when applicable, a read operation (Output enable)
  - 8 lanes for data to be sent in
  - 8 lanes for data to be collected
DRAM

- DRAM type memory: the signal is stored as a charge in a capacitor
  - No charge: 0 signal
  - Some charge: 1 signal

- The good: cheap, requires only one capacitor and one transistor

- The bad: capacitors leak, so the charge or lack of charge should be reinforced every so often ⇒ from where the name “dynamic” RAM
  - State of the capacitor should be refreshed every millisecond or so
  - Refreshing requires a small delay in memory accesses

- Is this delay incurred often? (first order approximation answer)
  - Given frequency at which memory is accessed, refreshing every millisecond means issues might appear once every million cycles
  - Turns out that 99% of memory cycles are useful; refresh operations consume 1% of DRAM memory cycles

[Patterson & H]→
SRAM vs. DRAM: wrap-up

- Order of the SRAM access time: 0.5ns
  - Expensive but fast
  - It’s mostly on chip
  - Needs no refresh

- Order of the DRAM access time: 50ns
  - Less expensive but slow
  - It’s mostly off chip
  - Higher capacity per unit area
  - Needs refresh every 10-100 ms
  - Sensitive to disturbances

- Limit case: a 100X speedup if you can work off the SRAM

<table>
<thead>
<tr>
<th></th>
<th>Transistors per bit</th>
<th>Access Time</th>
<th>Persistent?</th>
<th>Sensitive?</th>
<th>Price</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6</td>
<td>1X</td>
<td>Yes</td>
<td>No</td>
<td>100X</td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>No</td>
<td>Yes</td>
<td>1X</td>
<td>Main Memory</td>
</tr>
</tbody>
</table>
# Feature Comparison Between Memory Types

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>DRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed</strong></td>
<td>Very fast</td>
<td>Fast</td>
<td>Very slow</td>
</tr>
<tr>
<td><strong>Density</strong></td>
<td>Low</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td><strong>Endurance</strong></td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>Low</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td><strong>Refresh</strong></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Retention</strong></td>
<td>Volatile</td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td><strong>Mechanism</strong></td>
<td>Bi-stable Latch</td>
<td>Capacitor</td>
<td>Fowler-Nordheim tunneling</td>
</tr>
</tbody>
</table>
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Caches
Virtual Memory
Parallel Computing: Why, and Why Now?

September 16, 2013

"The Internet is a great way to get on the net."
[Former] US Senator Bob Dole
Before We Get Started...

- Last time
  - Pipelining
  - SRAM/DRAM

- Today
  - Brief discussion of memory: caches and main memory
  - Brief discussion of the Virtual Memory
  - Parallel Computing: Why?, and Why Now?

- Miscellaneous
  - First assignment, HW01, due on Monday at 11:59 PM
  - Second assignment posted on the course website later today
  - Read pages 28 through 56 of the primer available on the website
  - Contact Andrew Seidl aaseidl@wisc.edu if you haven’t got an Euler account
  - Make an early attempt to upload a file through Learn@UW. Check if it works for you
# Feature Comparison Between Memory Types

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>DRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Very fast</td>
<td>Fast</td>
<td>Very slow</td>
</tr>
<tr>
<td>Density</td>
<td>Low</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td>Endurance</td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Power</td>
<td>Low</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td>Refresh</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Retention</td>
<td>Volatile</td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Bi-stable Latch</td>
<td>Capacitor</td>
<td>Fowler-Nordheim tunneling</td>
</tr>
</tbody>
</table>
Cost and Speed Implications

- Since SRAM is expensive and bulkier, can’t have too much
  - Plagued by Space & Cost constraints

- Compromise:
  - Have some SRAM on-chip, making up what is called the “cache”
  - Have a lot of inexpensive DRAM off-chip, making up the “main memory”

- Hopefully your program has a low “average memory access time” by hitting the cache repeatedly instead of taking costly trips to main memory
Fallout: Memory Hierarchy

- You now have a “memory hierarchy”

- Simplest memory hierarchy:
  - Main Memory + One Cache (typically called L1 cache)

- Today’s memory architectures typically have deeper hierarchy: L1+L2+L3
  - L1 faster and smaller than L2
  - L2 faster and smaller than L3

- Note that all caches are typically on the chip
Example: Intel Chip Architecture

- Quad core Intel CPU die that illustrates L3 cache
- For Intel Core i7 975 Extreme, cache hierarchy is as follows
  - 32 KB L1 cache / core
  - 256 KB L2 (Instruction & Data) cache / core
  - 8 MB L3 (Instruction & Data) shared by all cores
Memory Hierarchy

- Memory hierarchy is deep:

Moving on to talk about caches
Cache Types

- Two main types of cache

- **Data** caches feed processor with data manipulated during execution
  - If processor would rely on data provided by main memory the execution would be pitifully slow
    - Processor Clock faster than the Memory Clock
    - Caches alleviate this memory pressure

- **Instruction** caches: used to store instructions
  - Much simpler to deal with compared to the data caches
    - Instruction use is much more predictable than data use

- In an ideal world, the processor would only communicate back and forth with the cache and avoid communication with the main memory
Split vs. Unified Caches

- Note that in the picture below L1 cache is split between data and instruction, which is typically the case
- L2 and L3 (when present) typically unified
How the Cache Works

- Assume simple setup with only one cache level L1

- Purpose of the cache: store for fast access a subset of the data stored in the main memory

- Data is moved at different resolutions between P ↔ C and between C ↔ M and
  - Between P and C: moved one word at a time
  - Between C and M: moved one block at a time (block called “cache line”)
Cache Hit vs. Cache Miss

- The processor typically agnostic about memory organization
- Middle man is the cache controller, which is an independent entity: it enables the “agnostic” attribute of the processor ↔ memory interaction
  - Processor requires data at some address
  - Cache Controller figures out if data is in a cache line
    - If yes: cache hit, processor served right away
    - If not: cache miss (data should be brought over from main memory → very slow)
- Difference between cache hit and cache miss:
  - Performance hit related to SRAM vs. DRAM memory access
More on Cache Misses…

- A cache miss refers to a failed attempt to read/write a piece of data from/to the cache, which results in a main memory access with much longer latency.

- There are three kinds of cache misses:
  - **Cache read miss from an instruction cache**: generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory.
  
  - **A cache read miss from a data cache**: usually causes less delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution.
  
  - **A cache write miss to a data cache**: generally causes the least delay, because the write can be queued and there are few limitations on the execution of subsequent instructions. The processor can continue unless the queue is full and then it has to stall for the write buffer to partially drain.
[It makes sense to ask this]

**Question:**

- Can you control what’s in the cache and anticipate future memory requests?
  - Typically not…
    - Any serious system has a hardware implemented cache controller with a mind of its own
  - There are ways to increase your chances of cache hits by designing software for high degree of memory access locality
  - Two flavors of memory locality:
    - Spatial locality
    - Temporal locality
Spatial and Temporal Locality

- **Spatial Locality for memory access by a program**
  - A memory access pattern characterized by bursts of repeated requests for data that is physically located within the same memory region
  - “Bursts” because this accesses should happen in a sufficiently short interval of time (otherwise the cache line gets evicted)

- **Temporal Locality for memory access by a program**
  - Idea: If you access a variable at some time, then you’ll probably keep accessing the same variable for a while
  - Example: have a for loop with some variables inside the loop → you keep accessing those variables as long as you loop
Cache Characteristics

- Size attributes: absolute cache size and cache line size
- Strategy for mapping of memory blocks to cache lines
- Cache line replacement algorithms
- Write-back policies

NOTE: these characteristics carry over and become more convoluted when dealing with multilevel cache hierarchies
The Concept of Virtual Memory
Motivating Questions/Issues

- Assumption: we are not talking about embedded systems, which are running alone on a processor and basically do not require an operating system to play the role of the middle man.

- Question 1: On a 32 bit machine, how come you can have 512MB of main memory yet allocate an array of 1 GB?

- Question 2: How can you compile a program on a Windows workstation with 2 GB of memory and run it later on a different laptop with 512 MB of memory?

- Question 3: How can several processes run seemingly at the same time on a processor with one thread?
The three questions raised on previous slide answered by the interplay between the compiler, the operating system (OS), and the execution model adopted by the processor.

When you compile a program there is no way to know where in the physical memory the code will get its data allocated.

- There are other “tenants” that inhabit the memory, and they are there before you get there.

The solution is for the code to be compiled and assumed to lead to a process that executes in a virtual world in which it has access to 4 GB of memory (on 32 bit systems).

- The “virtual world” is called the virtual memory space.
Virtual vs. Physical Memory

- Virtual memory: this nice and immaculate space of $2^{32}$ addresses (on 32 bit architectures) in which a process sees its data being placed, the instructions stored, etc.

- Physical memory: a busy place that hosts at the same time data and instructions associated with tens of applications running on the system
Anatomy of the Virtual Memory

**STACK segment**
- Stores a collection of frames, each associated with one function call
- A stack frame stores function parameters, return addresses, local variables, etc.
- Last-in-first-out (LIFO) structure; push/pop managed

**HEAP segment**
- Segment used when the program allocates memory dynamically, at run time
- Managed by the OS in response to function calls like malloc, free, etc.

**BSS segment**
- Stores uninitialized global and static variables

**DATA segment**
- Stores static variables and initialized global variables

**TEXT segment**
- Stores instructions associated with the program

**STACK OVERFLOW**
- If the top of the stack reaches beyond this logical address
- Cannot grow beyond this address

**Variable size**
- Can move this way [upon return of a function]
- Can move this way [upon a function call]

**Free memory**
- Can move up [upon call to malloc(), etc.]
- Can move down [upon call to free(), etc.]

**Virtual Address Space of a process**
- Lowest logical address [0x0000...]
- Highest logical address [0x ffff...]

Bottom of STACK

Top of STACK
The Anatomy of the Stack

- Function `bar` and associated stack frame

float bar(int a, float b)
{
    int initials[2];
    float t1, t2, t3;
    //...code here..
    //...no other variables..
    return t1;
}
The Virtual Memory.
The Page Table

- Virtual memory allows the processor to work in a virtual world in which each process, when run by the processor, seems to have exclusive access to a very large memory space

- For 32 bits: memory space is 4 GB big

- This virtual world is connected back to the physical memory through a Page Table
Anatomy of a Virtual Memory Address

- A virtual address has two parts: the page number, and the offset
Anatomy of a Virtual Memory Address

- A page of virtual memory corresponds to a frame of physical memory

- The size of a page (or frame, for that matter) is typically 4096 bytes

- $2^{12} = 4096$: 12 address bits are sufficient to relatively position each byte in a page
The Translation Process

- Example: imagine that your physical memory is 2 GB
- The physical address has 31 bits: $2^{31} = 2\text{GB}$
- Then the page table converts bits 12 through 31 of the virtual address into bits 12 through 30 of the physical address
Short Digression 1: The Unit of Address Resolution

- How many bits are available for data storage at each address?
- Example:
  - We have $2^{32}$ addresses that we can access
  - If each address points to a location that stores 8 bits (one byte) then we have 4 GB of addressable memory
  - However, if each address refers to a location that stores 2 bytes, we have 8 GB of addressable memory
- Intel and AMD CPUs: the unit of address resolution is 1 byte (8 bits)
- Consequence: the Intel 32 bit processors “see” a virtual memory space that can be 4 GB big
Short Digression 2: The 32 to 64 bit Migration

- If the architecture and OS have 32 bits to represent addresses, it means that $2^{32}$ addresses can be referenced.

- If the unit of address resolution is 1 byte, that means that the size of the virtual memory space can be 4 GB.

- This is hardly enough today when programs are very large and the amounts of data they manipulate can be staggering.

- This motivated the push towards having addresses represented using 64 bits: the memory space balloons to $2^{64}$ bytes, that is 16 times 1152921504606846976 bytes.
Short Digression 3: The 32 to 64 Bit Migration

- Note that a 64 bit architecture typically calls for two things:

- From a **hardware** perspective, the size of the registers, integer size, and word size is 64 bits.

- From a **software** perspective, the addresses are now 64 bits and therefore a program “operates” in a huge virtual memory space.
  - The operating system (OS) is the party managing the execution of a program in the 64 bit universe.
Comments on the Page Table
Preamble to TLB.

- The page table is the key ingredient that allows the translation of virtual addresses into physical addresses.

- Every single process executing on a processor and managed by the OS has its own page table.

- Page table is stored in main memory.
  - For a 32 bit operating system size of a page table can be up to 4 MB in size.
Comments on the Page Table. The TLB

- If Page Table stored in main memory it means that each address translation would require a trip to main memory
  - This would be very costly

- There is a “cache” for this translation process: TLB
  - Translation lookaside buffer: holds the translation of a small collection of virtual page numbers into frame IDs

- Best case scenario: the TLB leads to a hit and allows for quick translation
- Bad scenario: the TLB doesn’t have the required information cached and a trip to main memory is in order
- Worst scenario: the requested frame is not in main memory and a trip to secondary memory is in order
  - Called “page fault”
Illustration: The Role of the TLB

- A TLB is just like a cache
- A TLB miss leads to substantial overhead in the translation of an address
Memory Access: The Big Picture

- A simplified version of how a memory request is serviced presented below
Parallel Computing: Why, and Why Now?
Overview of Intel’s Haswell
Overview of NVIDIA’s Fermi

September 18, 2013

“In theory there is no difference between theory and practice.
In practice there is.”
Yogi Berra
Before We Get Started…

- Last time
  - Brief discussion of memory: caches and main memory
  - Brief discussion of the Virtual Memory

- Today
  - Parallel Computing: Why?, and Why Now?
  - The three walls to sequential computing

- Miscellaneous
  - Second assignment, HW02, due on Monday at 11:59 PM
  - Read pages 28 through 56 of the primer available on the website
  - HW submission policy will continue to be enforced as stated
Parallel Computing: Why? & Why Now?
The Argument in Today’s Lecture

- Sequential computing has been losing steam recently

- The immediate future seems to belong to parallel computing
Acknowledgements

- Material presented today includes content due to
  - Hennessy and Patterson (Computer Architecture, 4th edition)
  - John Owens, UC-Davis
  - Darío Suárez, Universidad de Zaragoza
  - John Cavazos, University of Delaware
  - Others, as indicated on various slides
  - I apologize if I included a slide and didn’t give credit where was due
CPU Speed Evolution

[log scale]

Courtesy of Elsevier: from Computer Architecture, Hennessy and Patterson, fourth edition
...we can expect very little improvement in serial performance of general purpose CPUs. So if we are to continue to enjoy improvements in software capability at the rate we have become accustomed to, we must use parallel computing. This will have a profound effect on commercial software development including the languages, compilers, operating systems, and software development tools, which will in turn have an equally profound effect on computer and computational scientists.

John L. Manferdelli, Microsoft Corporation Distinguished Engineer, leads the eXtreme Computing Group (XCG) System, Security and Quantum Computing Research Group
Three Walls to Serial Performance

- Memory Wall
- Instruction Level Parallelism (ILP) Wall
- Power Wall

Not necessarily walls, but increasingly steep hills to climb

http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/
Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between CPU and memory outside the CPU chip.

- Memory latency is a barrier to computer performance improvements
  - Current architectures have ever growing caches to improve the average memory reference time to fetch or write instructions or data.

- Memory Wall: due to latency and limited communication bandwidth beyond chip boundaries.
  - From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at 10%.
Memory Bandwidths
[typical embedded, desktop and server computers]
Memory Speed: Widening of the Processor-DRAM Performance Gap

- The processor: victim of its own success
  - So fast it left the memory behind
  - A system (CPU-Memory duo) can’t move as fast as you’d like (based on CPU top speeds) with a sluggish memory

- Plot on next slide shows on a *log* scale the increasing gap between CPU and memory

- The memory baseline: 64 KB DRAM in 1980

- Memory speed increasing at a rate of approx 1.07/year
  - However, processors improved
    - 1.25/year (1980-1986)
    - 1.52/year (1986-2004)
    - 1.20/year (2004-2010)
Memory Speed:
Widening of the Processor-DRAM Performance Gap

![Graph showing the widening gap between processor and memory performance from 1980 to 2010.](image)

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Memory Latency vs. Memory Bandwidth

- **Latency**: the amount of time it takes for an operation to complete
  - Measured in seconds
  - The utility “ping” in Linux measures the latency of a network
  - For memory transactions: send 32 bits to destination and back, measure how much time it takes → gives you latency

- **Bandwidth**: how much data can be transferred per second
  - You can talk about bandwidth for memory but also for a network (Ethernet, Infiniband, modem, DSL, etc.)

- **Improving Latency and Bandwidth**
  - The job of colleagues in Electrical Engineering
  - Once in a while, Materials Science colleagues deliver a breakthrough
  - Promising technology: optic networks and layered memory on top of chip
Memory Latency vs. Memory Bandwidth

- Memory Access Latency is significantly more challenging to improve as opposed to improving Memory Bandwidth

- Improving Bandwidth: add more “pipes”.
  - Requires more pins that come out of the chip for DRAM, for instance.
  - Adding more pins is not simple – very crowded real estate plus the technology is tricky

- Improving Latency: no easy answer here

- Analogy:
  - If you carry commuters with a train, add more cars to a train to increase bandwidth
  - Improving latency requires the construction of high speed trains
    - Very expensive
    - Requires qualitatively new technology (Elon Musk’s Hyperloop)
Latency vs. Bandwidth Improvements Over the Last 25 years

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
The 3D Memory Cube
[possible breakthrough?]

- Micron's Hybrid Memory Cube (HMC) features a stack of individual chips connected by vertical pipelines or “vias,” shown in the pic.
- IBM’s new 3-D manufacturing 32 nm technology, used to connect the 3D micro structure, will be the foundation for commercial production of the new memory cube.

- HMC prototypes clock in with bandwidth of 128 gigabytes per second (GB/s).
  - By comparison, current devices deliver roughly 15-25 GB/s.
- HMC also requires 70 percent less energy to transfer data.
- HMC offers a small form factor — just 10 percent of the footprint of conventional memory.

Memory Wall, Conclusions

[IMPORTANT ME759 SLIDE]

- Memory trashing is what kills execution speed

- Many times you will see that when you run your application:
  - You are far away from reaching top speed of the chip
    AND
  - You are at top speed for your memory
    - If this is the case, you are trashing the memory
    - Means that basically you are doing one or both of the following
      - Move large amounts of data around
      - Move data often

<table>
<thead>
<tr>
<th>Memory Access Patterns</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>To/From Registers</td>
<td>Golden</td>
</tr>
<tr>
<td>To/From Cache</td>
<td>Superior</td>
</tr>
<tr>
<td>To/From RAM</td>
<td>Trouble</td>
</tr>
<tr>
<td>To/From Disk</td>
<td>Salary cut</td>
</tr>
</tbody>
</table>
Instruction Level Parallelism (ILP)

- ILP: a relevant factor in reducing execution times after 1985

- The basic idea:
  - *Overlap execution* of independent instructions to improve overall performance
  - During *the same clock cycle* many instructions are being worked upon

- Two approaches to discovering ILP
  - Dynamic: relies on hardware to discover/exploit parallelism dynamically at run time
    - It is the dominant one in the market
  - Static: relies on compiler to identify parallelism in the code and leverage it (VLIW)

- Examples where ILP expected to improve efficiency
  ```
  for( int=0; i<1000; i++)
      x[i] = x[i] + y[i];
  1. e = a + b
  2. f = c + d
  3. g = e * f
  ```
ILP: Various Angles of Attack

- **Instruction pipelining**: the execution of multiple instructions can be partially overlapped; where each instruction is divided into series of sub-steps (termed: micro-operations)

- **Superscalar execution**: multiple execution units are used to execute multiple instructions in parallel

- **Out-of-order execution**: instructions execute in any order but without violating data dependencies

- **Register renaming**: a technique used to avoid data hazards and thus lead to unnecessary serialization of program instructions caused by the reuse of registers

- **Speculative execution**: allows the execution of complete instructions or parts of instructions before being sure whether this execution is required

- **Branch prediction**: used to avoid delays (termed: stalls). Used in combination with speculative execution.
How Microarchitecture Reflected into Execution

- Squeeze the most out of each cycle…
- Vertical axis: a summary of hardware assets
- Horizontal axis: time
The ILP Wall

- For ILP to make a dent, you need large blocks of instructions that can be [attempted to be] run in parallel

- Duplicate hardware speculatively executes future instructions before the results of current instructions are known, while providing hardware safeguards to prevent the errors that might be caused by out of order execution

- Branches must be “guessed” to decide what instructions to execute simultaneously
  - If you guessed wrong, you throw away that part of the result

- Data dependencies may prevent successive instructions from executing in parallel, even if there are no branches
The ILP Wall

- **ILP, the good:**
  - Existing programs enjoy performance benefits without any modification
  - Recompiling them is beneficial but entirely up to you as long as you stick with the same ISA (for instance, if you go from Pentium 2 to Pentium 4 you don’t have to recompile your executable)

- **ILP, the bad:**
  - Improvements are difficult to forecast since the “speculation” success is difficult to predict
  - Moreover, ILP causes a super-linear increase in execution unit complexity (and associated power consumption) without linear speedup.

- **ILP, the ugly:** serial performance acceleration using ILP plateauing because of these effects
The Power Wall

- Power, and not manufacturing, limits traditional general purpose microarchitecture improvements (F. Pollack, Intel Fellow)

- Leakage power dissipation gets worse as gates get smaller, because gate dielectric thicknesses must proportionately decrease

Adapted from F. Pollack (MICRO’99)
The Power Wall

- Power dissipation in clocked digital devices is related to the clock frequency and feature length imposing a natural limit on clock rates.

- Significant increase in clock speed without heroic (and expensive) cooling is not possible. Chips would simply melt.

- Clock speed increased by a factor of 4,000 in less than two decades:
  - The ability of manufacturers to dissipate heat is limited though...
  - Look back at the last five years, the clock rates are pretty much flat.

- Problem might be addressed one day by a Materials Science breakthrough.
Trivia

- AMD Phenom II X4 955 (4 core load)
  - 236 Watts

- Intel Core i7 920 (8 thread load)
  - 213 Watts

- Human Brain
  - 20 W
  - Represents 2% of our mass
  - Burns 20% of all energy in the body at rest
Conventional Wisdom in Computer Architecture

- Old: Power is free, Transistors expensive
- New: Power expensive, Transistors free
  (Can put more on chip than can afford to turn on)

- Old: Multiplies are slow, Memory access is fast
- New: Memory slow, multiplies fast [“Memory wall”]
  (400-600 cycles for DRAM memory access, 1 clock for FMA)

- Old: Increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)
- New: “ILP wall” diminishing returns on more ILP

- New: Power Wall + Memory Wall + ILP Wall = Brick Wall
  - Old: Uniprocessor performance 2X / 1.5 yrs
  - New: Uniprocessor performance only 2X / 5 yrs?
First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat.

[...] Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies.

[...] Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster further undercutting any gains that frequency increases might otherwise buy.
Summarizing It All…

- The sequential execution model is losing steam
- The bright spot: number of transistors per unit area going up and up
• OK, now what?
Moore’s Law

- 1965 paper: Doubling of the number of transistors on integrated circuits every two years
  - Moore himself wrote only about the density of components (or transistors) at minimum cost

- Increase in transistor count is also a rough measure of computer processing performance

Moore’s Law (1965)

- “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.”

“Cramming more components onto integrated circuits” by Gordon E. Moore, Electronics, Volume 38, Number 8, April 19, 1965
The Ox vs. Chickens Analogy

Seymour Cray: "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

- Chicken is gaining momentum nowadays:
  - For certain classes of applications, you can run many cores at lower frequency and come ahead at the speed game

- Example:
  - Scenario One: one-core processor w/ power budget W
    - Increase frequency by 20%
      - Substantially increases power, by more than 50%
      - But, only increase performance by 13%
  
  - Scenario Two: Decrease frequency by 20% with a simpler core
    - Decreases power by 50%
    - Can now add another dumb core (one more chicken…)
Intel’s Vision: Evolutionary Configurable Architecture

- **Many-core array**
  - CMP with 10s-100s low power cores
  - Scalar cores
  - Capable of TFLOPS+
  - Full System-on-Chip
  - Servers, workstations, embedded...

- **Scalar plus many core** for highly threaded workloads
- **Multi-core array**
  - CMP with ~10 cores

- **Dual core**
  - Symmetric multithreading

CMP = “chip multi-processor”

Presentation Paul Petersen, Sr. Principal Engineer, Intel
Intel Roadmap

- 2013 – 22 nm
- 2015 – 14 nm
- 2017 – 10 nm
- 2019 – 7 nm
- 2021 – 5 nm
- 2023 – ??? (your turn)
Old School

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, just wait and run on much faster sequential computer
- Less than linear scaling for a multiprocessor is failure

New School

- Processors parallelism is primary method of performance improvement
- Nobody is building one processor per chip. This marks the end of the La-Z-Boy programming era
- Given the switch to parallel hardware, even sub-linear speedups are beneficial as long as you beat the sequential

Slide Source: Berkeley View of Landscape
Implications in the Software Business

- “Parallelism for Everyone”
- Parallelism changes the game
  - A large percentage of people who provide applications are going to have to care about parallelism in order to match the capabilities of their competitors.

competitive pressures = demand for parallel applications
Moving Into Parallelism...
From Simple to Complex: Part 1

- The von Neumann architecture
From Simple to Complex: Part 2

- The architecture of the early to mid 1990s
  - Pipelining was king
From Simple to Complex: Part 3

- The architecture of late 1990s, early 2000s
  - ILP galore
Two Examples of Parallel HW

- Intel Haswell
  - Multicore architecture

- NVIDIA Fermi
  - Large number of scalar processors ("shades")
Intel Haswell

- June 2013
- 22 nm technology
- 1.4 billion transistors
- 4 cores, hyperthreaded
- Integrated GPU
- System-on-a-chip design
Intel Haswell: Front End and Back End

- A high level organization:
  - Decoding
  - Scheduling
  - Execution
Intel Haswell: Overall Perspective

- At the right: complete schematic of microarchitecture
- More info: see online primer
- Good overview provided here: http://www.realworldtech.com/haswell-cpu/
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP)
- L1 cache
- L2 cache
- 6 GB of global memory
- Operates at low clock rate
- High bandwidth (close to 200 GB/s)
Fermi: 30,000 Feet Perspective

- Lots of ALU (green), not much of CU
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory
A Fermi Core (or SM – Streaming Multiprocessor)
Overview of NVIDIA’s Fermi
Big Iron HPC Alternatives
Computing on the GPU

September 20, 2013

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?”
Brian Kernighan
Before We Get Started…

- Last time
  - Parallel Computing: Why?, and Why Now?
  - The three walls to sequential computing

- Today
  - Overview of Fermi
  - Parallel computing on large supercomputers
  - Start segment on computing on the GPU

- Miscellaneous
  - Second assignment, HW02, due on Monday at 11:59 PM
  - Read pages 28 through 56 of the primer available on the website
  - HW submission policy will continue to be enforced as stated
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- L1 cache
- L2 cache
- 6 GB of global memory
- Operates at low clock rate
- High bandwidth (close to 200 GB/s)
Fermi: 30,000 Feet Perspective

- Lots of ALU (green), not much of CU
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory
“Big Iron” Parallel Computing
Euler: CPU/GPU Heterogeneous Cluster
~ Hardware Configuration ~

Legend, Connection Type:
- Gigabit Ethernet
- 4x QDR Infiniband

File Server Architecture
- CPU Intel Xeon 5620
- RAM 16 GB DDR3
- Infiniband HCA
- RAID 6
- 24x 2TB Hard Disks

CPU/GPU Node Architecture
- CPU 0
  - Intel Xeon 5520
  - Hard Disk
- CPU 1
  - Intel Xeon 5520
  - Infiniband HCA
- GPU 0
  - GPU 1
  - GPU 2
  - GPU 3
- RAM 48 GB DDR3
- GTX 480
  - 1.5GB RAM
  - 448 Cores
  - PCIEx16 2.0

AMD Node Architecture
- CPU 0
  - AMD Opteron 6276
- CPU 1
  - AMD Opteron 6276
- CPU 2
  - AMD Opteron 6276
- CPU 3
  - AMD Opteron 6276
- RAM 128 GB DDR3
- Infiniband HCA
- SSD
Euler, in reality...
Overview of Large Multiprocessor Hardware Configurations ("Big Iron")

- Larger multiprocessors
  - Shared address space
    - Symmetric shared memory (SMP)
      - Examples: IBM eserver, SUN Sunfire
    - Distributed shared memory (DSM)
  - Distributed address space
    - Commodity clusters: Beowulf and others
    - Custom cluster
      - Cache coherent: ccNUMA
        - SGI Origin/Altix
      - Noncache coherent: Cray T3E, X1
      - Uniform cluster:
        - IBM BlueGene
      - Constellation cluster of DSMs or SMPs
        - SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.
Some Nomenclature…

- Shared addressed space: when you invoke address “0x0043fc6f” on one machine and then invoke “0x0043fc6f” on a different machine they actually point to the same global memory space
  - Issues: memory coherence
    - Fix: software-based or hardware-based

- Distributed addressed space: the opposite of the above

- Symmetric Multiprocessor (SMP): you have one machine that shares amongst all its processing units a certain amount of memory (same address space)
  - Mechanisms should be in place to prevent data hazards (RAW, WAR, WAW). Brings back the issue of memory coherence

- Distributed shared memory (DSM):
  - Also referred to as distributed global address space (DGAS)
  - Although physically memory is distributed, it shows as one uniform memory
  - Memory latency is highly unpredictable
Example

- Distributed-memory multiprocessor architecture (Euler, for instance)
Comments, distributed-memory multiprocessor architecture

- Basic architecture consists of nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes.

- Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

- Popular interconnection network: Mellanox and Qlogic InfiniBand
  - Bandwidth range: 1 through 50 Gb/sec
  - Latency: in the microsecond range (approx. 1E-6 seconds)
  - Requires special network cards: HCA – “Host Channel Adaptor”

- InfiniBand offers point-to-point bidirectional serial links intended for the connection of processors with high-speed peripherals such as disks.
  - Basically, a protocol and implementation for communicating data very fast
  - It supports several signaling rates and, as with PCI Express, links can be bonded together for additional throughput
  - Similar technologies: Fibre Channel, PCI Express, Serial ATA, etc.
  - Euler: uses 4X Infiniband QDR for 40 Gb/sec bandwidth
Example, SMP

[This is not “Big Iron”, rather a desktop nowadays]

- Shared-Memory Multiprocessor Architecture

Usually SRAM

Usually DRAM

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Comments, SMP Architecture

- Multiple processor-cache subsystems share the same physical off-chip memory

- Typically connected to this off-chip memory by one or more buses or a switch

- Key architectural property: uniform memory access (UMA) time to all of memory from all the processors
  - This is why it’s called symmetric
Examples…

- **Shared-Memory**
  - Intel Xeon Phi available as of 2012
    - Packs 61 cores, which are on the basic (unsophisticated) side
  - AMD Opteron 6200 Series (16 cores: Opteron 6276) – Bulldozer architecture
  - Sun Niagara

- **Distributed-Memory**
  - IBM BlueGene/L
  - Cell (see [http://users.ece.utexas.edu/~adnan/vlsi-07/hofstee-cell.ppt](http://users.ece.utexas.edu/~adnan/vlsi-07/hofstee-cell.ppt))
### Big Iron: Where Are We Today?

[Info lifted from Top500 website: http://www.top500.org/]

<table>
<thead>
<tr>
<th>RANK</th>
<th>NAME</th>
<th>SPECIFICATIONS</th>
<th>SITE</th>
<th>COUNTRY</th>
<th>CORES</th>
<th>R_MAX_PLOP/S</th>
<th>POWER_MW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tianhe-2 (Milkyway-2)</td>
<td>NUDT, Intel Ivy Bridge (12C, 2.2 GHz) &amp; Xeon Phi (57C, 1.1 GHz), Custom interconnect</td>
<td>NUDT</td>
<td>China</td>
<td>3,120,000</td>
<td>33.9</td>
<td>17.8</td>
</tr>
<tr>
<td>2</td>
<td>Titan</td>
<td>Cray XK7, Opteron 6274 (16C, 2.2 GHz) + Nvidia Kepler (14C, 732 GHz), Custom interconnect</td>
<td>DOE/SC/ORNL</td>
<td>USA</td>
<td>560,640</td>
<td>17.6</td>
<td>8.3</td>
</tr>
<tr>
<td>3</td>
<td>Sequoia</td>
<td>IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
<td>1,572,864</td>
<td>17.2</td>
<td>7.9</td>
</tr>
<tr>
<td>4</td>
<td>K computer</td>
<td>Fujitsu SPARC64 VIIifx (8C, 2.0GHz), Custom interconnect</td>
<td>RIKEN AICS</td>
<td>Japan</td>
<td>705,024</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td>5</td>
<td>Mira</td>
<td>IBM BlueGene/Q, Power BQC (16C, 1.60 GHz), Custom interconnect</td>
<td>DOE/SC/ANL</td>
<td>USA</td>
<td>786,432</td>
<td>8.16</td>
<td>3.95</td>
</tr>
</tbody>
</table>

#### PERFORMANCE DEVELOPMENT

![Diagram showing performance development over time](image-url)
Abbreviations/Nomenclature

- MPP – Massively Parallel Processing
- Constellation – subclass of cluster architecture envisioned to capitalize on data locality
- MIPS – “Microprocessor without Interlocked Pipeline Stages”, a chip design of the MIPS Computer Systems of Sunnyvale, California
- SPARC – “Scalable Processor Architecture” is a RISC instruction set architecture developed by Sun Microsystems (now Oracle) and introduced in mid-1987
- Alpha - a 64-bit reduced instruction set computer (RISC) instruction set architecture developed by DEC (Digital Equipment Corporation was sold to Compaq, which was sold to HP) – adopted by Chinese chip manufacturer (see primer)
Short Digression [first take]:
What is a MPP?

- Large-scale computer system that
  - Uses commodity microprocessors in processing nodes
  - Uses physically distributed memory nodes
  - Uses custom-designed interconnect w/ high bandwidth & low latency
  - Can be scaled up to hundreds or more processors

- Examples:
  - Intel ASCI TeraFLOPS, IBM SP2, Cray T3D and T3E (also DSM machines), Intel Paragon

[Zhiwei Xu and Kai Hwang]→
Short Digression [second take]:

What is a MPP?

- A very large-scale comp. sys. with commodity processing nodes interconnected with a high-speed low-latency interconnect
- Memories are physically distributed
- Nodes often run a microkernel
- Contains one host monolithic OS
- There are overlaps among MPPs, clusters, and SMPs
Short Digression [third take]:
What is a MPP?

- Uses commercial microprocessors at each node
- Physically distributed memory across nodes
- High bandwidth communication network w/ nearly zero latency
- Scales to hundreds and even thousands of processing nodes
- Asynchronous execution (vs. syn. as in SIMD)
- Treating distributed memory as an unshared (vs. shared as in DSM) resource

- Examples: Intel Paragon, TFLOP

[R. Jenkins]→
How is the speed measured to put together the Top500?

- Basically reports how fast you can solve a dense linear system
Some Trends...

- **Consequence of Moore’s law**
  - Transition from a speed-based compute paradigm to a concurrency-based compute paradigm (from few oxen to many chickens)

- **Amount of power for supercomputers is a showstopper**
  - Example:
    - Exascale Flops/s rate: reach it by 2018
    - Budget constraints: must be less than $200 million
    - Power constraints: must require less than 20 MW hour
  - Putting things in perspective:
    - China’s fastest supercomputer in 2011: 4.04 Mwatts for 2.57 Petaflop/s
    - Oak Ridge Jaguar’s (about 2012): 7.0 Mwats for 1.76 Petaflop/s
    - Faster machine for less power: took advantage of GPU computing
Relation Between HPC and “Big Iron”

- What people understand through HPC is typically the use of “big iron” machines

- ME759 called “HPC for Engineering Applications”
  - Somewhat of a misnomer then, since we’ll spend a lot of time discussing GPU computing (which is not “big iron”)

- Better name for class would have been “Parallel Computing for Engineering Applications”

- Oh well…
Flynn’s Taxonomy of Architectures

- There are several ways to classify architectures (we just saw on based on how memory is organized/accessed)

- Below, classified based on how instructions are executed in relation to data

  - SISD - Single Instruction/Single Data
  - SIMD - Single Instruction/Multiple Data
  - MISD - Multiple Instruction/Single Data
  - MIMD - Multiple Instruction/Multiple Data
Single Instruction/Single Data Architectures

Your desktop, before the spread of dual core CPUs

Flavors of SISD

Instructions:
Single Instruction/Multiple Data Architectures

Processors that execute same instruction on multiple pieces of data: NVIDIA GPUs

Single Instruction/Multiple Data [Cntd.]

- Each core runs the same set of instructions on different data
- Examples:
  - Graphics Processing Unit (GPU): processes pixels of an image in parallel
  - CRAY’s vector processor, see image below

Slide Source: Klimovitski & Macri, Intel
SISD versus SIMD

Writing a compiler for SIMD architectures is difficult (inter-thread communication complicates the picture...)

Slide Source: ars technica, Peakstream article
Multiple Instruction/Single Data

Not useful, not aware of any commercial implementation...
Multiple Instruction/Multiple Data

As of 2006, all the top 10 and most of the TOP500 supercomputers were based on a MIMD architecture.

Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- High Performance Computing
  - Topic of interest in this class
  - The idea: run one executable as fast as you can
    - Might spend one month running one DFT job or a week on a CFD job…

- High Throughput Computing
  - The idea: run as many applications as you can, possibly at the same time on different machines
  - Example: bone analysis in ABAQUS
    - You have uncertainty in the length of the bone (20 possible lengths) in the material of the bone (10 values for Young’s modulus) in the loading of the bone (50 force values with different magnitude/direction). Grand total: 10,000 ABAQUS runs
    - We have 1400 workstations hooked up together on-campus -> use Condor to schedule the 10,000 independent ABAQUS jobs and have them run on scattered machines overnight
  - Example: folding@home – volunteer your machine to run a MD simulation when it’s idle
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- **High Performance Computing**
  - Usually one cluster (e.g. Euler) or one massively parallel machine (e.g. IBM Blue Gene or Cray) that is dedicated to running one large application that requires a lot of memory, a lot of compute power, and a lot of communication
    - Example: each particle in a MD simulation requires (due to long range electrostatic interaction) to keep track of a large number of particles that it interacts with. Needs to query and figure out where these other particles are at any time step of the numerical integration
  - What is crucial is the interconnect between the processing units
    - Typically some fast dedicated interconnect (e.g. InfiniBand), which operates at 40 GB/s
      - Euclid@UW-Madison: 1 GB/s Ethernet, Bluewaters@UIU/C: 100 GB/s, Tianhe-I claims double the speed of Infiniband
  - Typically uniform hardware components: e.g. 100,000 Intel Xeon 5520, etc.
  - Comes at a premium $$$
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- High Throughput Computing
  - Usually a collection of heterogeneous compute resources linked through a slow connection, most likely Ethernet
    - Example: 120 Windows workstations in the CAE labs (all sorts of machines, some new, some old)
  
  - When CAE machine 58 runs an ABAQUS bone simulation there is no communication needed with CAE machine 83 that runs a different ABAQUS scenario
  
  - Don’t need to spend any money, you can piggyback on resources that are willing to make themselves available
  
  - Very effective to run Monte Carlo type analyses
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- You can do HPC on a configuration that has slow interconnect
  - It will run very very slow…

- You can do HTC on an IBM Blue Gene
  - You need to have the right licensing system in place to “check out” 10,000 ABAQUS licenses
  - You will use the processors but will waste the fast interconnect that made the machine expensive in the first place

- University of Wisconsin-Madison well known due to the pioneering work in the area of HTC done by Professor Livny in CS
  - UW-Madison solution for HTC: Condor, used by a broad spectrum of organizations from academia and industry
  - Other commercial solutions now available for HTC: PBSWorks, form Altair
  - Google and Amazon are heavily invested in the HTC idea

- The line between HPC and HTC is blurred when it comes to cloud computing
  - Cloud computing: you rely on hardware resources made available by a third party. The solution of choice today for HTC. If the machines in the cloud linked by fast interconnect one day might consider running HPC jobs there as well…
Amdahl's Law


“A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude”

- Let $r_s$ capture the amount of time that a program spends in components that can only be run sequentially
- Let $r_p$ capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that $r_s$ and $r_p$ are normalized, so that $r_s + r_p = 1$
- Let $n$ be the number of threads used to parallelize the part of the program that can be executed in parallel
- The “best case scenario” speedup $S$ is

$$S = \frac{T_{old}}{T_{new}} = \frac{r_s + r_p}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}$$
Amdahl’s Law

[Cntd.]

- Sometimes called the law of diminishing returns

- In the context of parallel computing used to illustrate how going parallel with a part of your code is going to lead to overall speedups

- The art is to find for the same problem an algorithm that has a large $r_p$
  - Sometimes requires a completely different angle of approach for a solution

- Nomenclature
  - Algorithms for which $r_p=1$ are called “embarrassingly parallel”
Example: Amdahl's Law

- Suppose that a program spends 60% of its time in I/O operations, pre and post-processing
- The rest of 40% is spent on computation, most of which can be parallelized
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem. What is the maximum amount of speedup that you can expect given this investment?
- Asymptotically, what is the maximum speedup that you can ever hope for?
A Word on “Scaling”
[important to understand]

- **Algorithmic Scaling** of a solution algorithm
  - You only have a mathematical solution algorithm at this point
  - Refers to how the effort required by the solution algorithm scales with the size of the problem
  - Examples:
    - Naïve implementation of the N-body problem scales like $O(N^2)$, where $N$ is the number of bodies
    - Sophisticated algorithms scale like $O(N \cdot \log N)$
    - Gauss elimination scales like the cube of the number of unknowns in your linear system

- **Implementation Scaling** on a certain architecture
  - **Intrinsic Scaling**: how the wall-clock run time changes with an increase in the size of the problem
  - **Strong Scaling**: how the wall-clock run time changes when you increase the processing resources
  - **Weak Scaling**: how the wall-clock run time changes when you increase the problem size but also the processing resources in a way that basically keeps the ration of problem size/processor constant
  - Relative relevance: strong and intrinsic more relevant than weak

- A thing you should worry about: is the Intrinsic Scaling similar to the Algorithmic Scaling?
  - If Intrinsic Scaling significantly worse than Algorithmic Scaling:
    - You might have an algorithm that thrashes the memory badly, or
    - You might have a sloppy implementation of the algorithm
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Computing on the GPU
CUDA and GPU Programming Model
Execution Configuration

September 23, 2013

“If you don’t want to be replaced by a computer, don’t act like one.”
Arno Penzias
Before We Get Started…

- Last time
  - Overview of Fermi
  - Parallel computing on large supercomputers

- Today
  - General discussion, computing on the GPU
  - The CUDA execution model

- Miscellaneous
  - Second assignment, HW02, due tonight at 11:59 PM
  - Third assignment, HW03 posted later today
  - Read pages 28 through 56 of the primer available on the website
  - HW submission policy will continue to be enforced as stated
End: Intro Part of ME759

Beginning: GPU Computing, CUDA Programming Model
Here’s where we are.

- Covered really fast a couple of hardware and micro-architecture aspects that are relevant to writing software
  - From transistor to CPU
  - From C code to machine instructions
  - How machine instructions are processed (FDX cycle)
  - Concepts related to the memory hierarchy
  - The concept of virtual memory
  - Instruction Level Parallelism (ILP)
  - The microarchitecture of Intel’s Haswell and NVIDIA’s Fermi
  - Big Iron HPC

- Moving on to GPU computing, present in more detail
Acknowledgements

- Many slides herein include material developed at the University of Illinois Urbana-Champaign by Professor W. Hwu and Adjunct Professor David Kirk (the latter also former Chief Scientist at NVIDIA).
  - Slides that include material produced by professors Hwu and Kirk contain a HK-UIUC logo in the lower left corner of the slide

- Several other slides are lifted from other sources as indicated along the way
Why Discuss GPU Computing?

- It’s fast for a variety of jobs
  - Really good for data parallelism (another way of saying SIMD)

- It’s cheap to get one ($120 to $480)
  - High end GPUs for Scientific Computing are more like $3000

- GPUs are everywhere
  - Chances are you have one or at least have easy access to one
Why GPU computing in ME759?

- GPU computing is not quite High Performance Computing (HPC)
  - However, it shares with HPC the important aspect that they both draw on parallel programming
  - A bunch of GPUs can together lead to a HPC cluster, see example of Tianhe-I, the fastest supercomputer in the world in early 2011

- GPUs are called sometimes accelerators or co-processors
  - Complement the capability of the CPU core[s]

- GPU proved very useful in computing collision detection, image processing, N-body problems, CFD, FFT, DFT, etc.

- More than 100 million NVIDIA GPU cards in use today
Layout of Typical Hardware Architecture

CPU (the “host”)

GPU w/ local DRAM (the “device”)

Wikipedia
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers

- Kepler K20X delivers 1.515 Tflops in double precision

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - “Extended C”
Bandwidth in a CPU-GPU System

NOTE: The width of the black lines is proportional to the bandwidth.
GPU vs. CPU – Memory Bandwidth

[GB/sec]
**CPU2GPU Transfer Issues: PCI-Express Latency**

- Relevant since host-device communication done over PCI-Express bus

---

### Table 1: Short Packet Reads

<table>
<thead>
<tr>
<th></th>
<th>PCI Express</th>
<th>HyperTransport</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S&amp;F C-T</td>
<td>S&amp;F C-T</td>
</tr>
<tr>
<td><strong>Latency (ns)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max Payload setting</strong></td>
<td>128 256</td>
<td>64 64</td>
</tr>
<tr>
<td>Number of request packets</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td><strong>Read Request</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx Application</td>
<td>15 6</td>
<td>12 6</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>15 15</td>
<td>12 12</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>20 20</td>
<td>6 6</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>30 30</td>
<td>8 8</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>15 15</td>
<td>12 12</td>
</tr>
<tr>
<td>Rx Application</td>
<td>15 6</td>
<td>12 6</td>
</tr>
<tr>
<td><strong>Fabric + DRAM cont. + DRAM (open)</strong></td>
<td>51 51</td>
<td>51 51</td>
</tr>
<tr>
<td><strong>Read Completion</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx Application (builds response packet)</td>
<td>18 6</td>
<td>12 6</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>18 15</td>
<td>12 9</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>20 20</td>
<td>6 6</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>30 30</td>
<td>8 8</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>18 18</td>
<td>12 12</td>
</tr>
<tr>
<td><strong>Total to get 1st byte of 1st packet back</strong></td>
<td>265 323</td>
<td>165 144</td>
</tr>
<tr>
<td>Rx Appl. (waits for all bytes @ link speed)</td>
<td>8 8</td>
<td>3 3</td>
</tr>
<tr>
<td><strong>TOTAL: Source→Link→CPU→Link→Sink</strong></td>
<td>273 240</td>
<td>168 147</td>
</tr>
</tbody>
</table>

---

### Table 2: Long Packet Reads

<table>
<thead>
<tr>
<th></th>
<th>PCI Express</th>
<th>HyperTransport</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S&amp;F C-T</td>
<td>S&amp;F C-T</td>
</tr>
<tr>
<td><strong>Latency (ns)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Max Payload setting</strong></td>
<td>128 256</td>
<td>64 64</td>
</tr>
<tr>
<td>Number of request packets</td>
<td>16 8</td>
<td>32 32</td>
</tr>
<tr>
<td><strong>Read Request</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx Application</td>
<td>15 6</td>
<td>12 6</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>15 15</td>
<td>12 12</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>20 20</td>
<td>6 6</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>30 30</td>
<td>8 8</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>15 15</td>
<td>12 12</td>
</tr>
<tr>
<td>Rx Application</td>
<td>15 6</td>
<td>12 6</td>
</tr>
<tr>
<td><strong>Fabric + DRAM cont. + DRAM (open)</strong></td>
<td>51 51</td>
<td>51 51</td>
</tr>
<tr>
<td><strong>Read Completion</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx Application (builds response packet)</td>
<td>63 6</td>
<td>33 6</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>63 15</td>
<td>33 9</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>20 20</td>
<td>6 6</td>
</tr>
<tr>
<td>SerDes + PMA + PCS + MAC</td>
<td>30 30</td>
<td>8 8</td>
</tr>
<tr>
<td>Data Link + Transaction Layers</td>
<td>63 111</td>
<td>33 33</td>
</tr>
<tr>
<td><strong>Total to get 1st byte of 1st packet back</strong></td>
<td>400 325</td>
<td>228 165</td>
</tr>
<tr>
<td>Rx Appl. (waits for all bytes @ link speed)</td>
<td>608 560</td>
<td>411 411</td>
</tr>
<tr>
<td><strong>TOTAL: Source→Link→CPU→Link→Sink</strong></td>
<td>1008 885</td>
<td>630 576</td>
</tr>
</tbody>
</table>

---

Comparison: Latency, DRAM Memory Access

<table>
<thead>
<tr>
<th>Year of introduction</th>
<th>Chip size</th>
<th>Slowest DRAM (ns)</th>
<th>Fastest DRAM (ns)</th>
<th>Column access strobe (CAS)/data transfer time (ns)</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64K bit</td>
<td>180</td>
<td>150</td>
<td>75</td>
<td>250</td>
</tr>
<tr>
<td>1983</td>
<td>256K bit</td>
<td>150</td>
<td>120</td>
<td>50</td>
<td>220</td>
</tr>
<tr>
<td>1986</td>
<td>1M bit</td>
<td>120</td>
<td>100</td>
<td>25</td>
<td>190</td>
</tr>
<tr>
<td>1989</td>
<td>4M bit</td>
<td>100</td>
<td>80</td>
<td>20</td>
<td>165</td>
</tr>
<tr>
<td>1992</td>
<td>16M bit</td>
<td>80</td>
<td>60</td>
<td>15</td>
<td>120</td>
</tr>
<tr>
<td>1996</td>
<td>64M bit</td>
<td>70</td>
<td>50</td>
<td>12</td>
<td>110</td>
</tr>
<tr>
<td>1998</td>
<td>128M bit</td>
<td>70</td>
<td>50</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>256M bit</td>
<td>65</td>
<td>45</td>
<td>7</td>
<td>90</td>
</tr>
<tr>
<td>2002</td>
<td>512M bit</td>
<td>60</td>
<td>40</td>
<td>5</td>
<td>80</td>
</tr>
<tr>
<td>2004</td>
<td>1G bit</td>
<td>55</td>
<td>35</td>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>2006</td>
<td>2G bit</td>
<td>50</td>
<td>30</td>
<td>2.5</td>
<td>60</td>
</tr>
</tbody>
</table>

*Figure 5.13 Times of fast and slow DRAMs with each generation.* (Cycle time is defined on page 310.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
CPU vs. GPU – Flop Rate (GFlops)

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Single Precision
- Double Precision

GFlop/Sec

2003 2004 2005 2006 2007 2008 2009 2010
More Up-to-Date, DP Figures…

Source: Revolutionizing High Performance Computing / Nvidia Tesla
What is the GPU so Fast?

- The GPU is specialized for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
  - More transistors can be devoted to data processing rather than data caching and control flow
  - Where are GPUs good: high arithmetic intensity (the ratio between arithmetic operations and memory operations)

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation
<table>
<thead>
<tr>
<th>Key Parameters</th>
<th>GPU – NVIDIA Tesla C2050</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Cores</strong></td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>3 GB</td>
<td>- 32 KB L1 cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 256 KB L2 (I&amp;D)cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 8 MB L3 (I&amp;D) shared by all cores</td>
</tr>
<tr>
<td><strong>Clock speed</strong></td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td><strong>Memory bandwidth</strong></td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td><strong>Floating point operations/s</strong></td>
<td><strong>515 x 10^9</strong> Double Precision</td>
<td><strong>70 x 10^9</strong> Double Precision</td>
</tr>
</tbody>
</table>
IBM BlueGene/L

- Entry model: 1024 dual core nodes
- 5.7 Tflop/s
- Linux OS

- Dedicated power management solution
- Dedicated IT support
- Decent options for productivity tools (debugging, profiling, etc.)
  - TotalView

- Price (2007): $1.4 million
When Are GPUs Good?

- Ideally suited for data-parallel computing (SIMD)

- Moreover, you want to have high arithmetic intensity
  - Arithmetic intensity: ratio of arithmetic operations to memory operations

- Example: quick back-of-the-envelope computation to illustrate the crunching number power of a modern GPU
  - Suppose it takes 4 microseconds (4E-6) to launch a kernel (more about this later…)
  - Suppose you own a 1 Tflops (1E12) Fermi-type GPU and use to add (in 4 cycles) floats
  - Then, you have to carry out about 1 million floating point ops on the GPU to break even with the amount of time it took you to invoke execution on the GPU in the first place
When Are GPUs Good?

[Cntd.]

- Another quick way to look at it:
  - Your 1 Tflops GPU needs a lot of data to keep busy and reach that peak rate
  - For instance: assume that you want to add *different* numbers and reach 1 Tflops: $1 \times 10^{12}$ ops/second…
  - You need to feed $2 \times 10^{12}$ operands per second…
  - If each number is stored using 4 bytes (float), then you need to fetch $2 \times 10^{12} \times 4$ bytes in a second. This is $8 \times 10^{12}$ B/s, which is 8 TB/s…
  - The memory bandwidth on the GPU is in the neighborhood of 0.15 TB/s, about 50 times less than what you need (and you haven’t taken into account that you probably want to send back the outcome of the operation that you carry out)

- Here’s a set of rules that you need to keep in mind before going further…
  - GET THE DATA ON THE GPU AND KEEP IT THERE
  - GIVE THE GPU ENOUGH WORK TO DO
  - FOCUS ON DATA REUSE WITHIN THE GPU TO AVOID MEMORY BANDWIDTH LIMITATIONS

Rules suggested by Rob Farber
GPU Computing – The Basic Idea

- GPU, going beyond graphics:
  - The GPU is connected to the CPU by a reasonable fast bus (8 GB/s is typical today)
  - The idea is to use the GPU as a co-processor
    - Farm out big parallel jobs to the GPU
    - CPU stays busy with the control of the execution and “corner” tasks
    - You have to copy data down into the GPU, and then fetch results back
      - Ok if this data transfer is overshadowed by the number crunching done using that data (remember Amdahl’s law…)
What is GPGPU?

[A Bit of History]

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

- Applications – see http://GPGPU.org
  - Game effects, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Shaders
[A Bit of History]

- A shader: set of software instructions mostly used to produce rendering effects on graphics hardware with a good degree of flexibility

- Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline
  - Represent a set of instructions executed by a GPU thread

- Shader-programming replaced the fixed-function pipeline that allowed only pre-canned common geometry transformation and pixel-shading functions

- Shaders enable customized effects
  - Vertex shader
  - Geometry shader
  - Pixel shader
GPGPU Constraints of the Past
[A Bit of History]

- Dealing with graphics API
  - Working with the corner cases of the graphics API

- Addressing modes
  - Limited texture size/dimension

- Shader capabilities
  - Limited outputs

- Instruction sets
  - Lack of Integer & bit ops

- Communication limited
  - Between pixels
  - Only gather (can read data from other pixels), but no scatter (can only write to one pixel)

Summing Up: Mapping computation problems to graphics rendering pipeline was tedious…
CUDA: Making the GPU Tick...

- “Compute Unified Device Architecture” – freely distributed by NVIDIA
- When introduced it eliminated the constraints associated with GPGPU
- It enables a general purpose programming model
  - User kicks off batches of threads on the GPU to execute a function (kernel)
- Targeted software stack
  - Scientific computing oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute, graphics free, API
  - Explicit GPU memory management
CUDA Programming Model: GPU as a Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory, or global memory in CUDA parlance)
  - Runs many threads in parallel

- Data-parallel portions of an application run on the device as kernels which are executed in parallel by many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few heavy ones
Fermi: Quick Facts

- Lots of ALU (green), not much of CU
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- 64 KB L1 cache
- 768 KB L2 uniform cache (shared by all SMs)
- Up to 6 GB of global memory
- Operates at several clock rates
  - Memory
  - Scheduler
  - Shader (SP)
- High memory bandwidth
  - Close to 200 GB/s
GPU Processor Terminology

- GPU is a SIMD device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that it is assigned to handle
  - The NVIDIA calls this model SIMT (single instruction multiple thread)

- The number crunching power comes from a vertical hierarchy:
  - A collection of Streaming Multiprocessors (SMs)
  - Each SM has a set of 32 Scalar Processors (SPs)

- The quantum of scalability is the SM
  - The more $ you pay, the more SMs you get inside your GPU
  - Fermi can have up to 16 SMs on one GPU card
Compute Capability [of a Device] vs. CUDA Version

- “Compute Capability of a Device” refers to **hardware**
  - Defined by a major revision number and a minor revision number

- Example:
  - Tesla C1060 is compute capability 1.3
  - Tesla C2050 is compute capability 2.0
  - Fermi architecture is capability 2 (on Euler now)
  - Kepler architecture is capability 3 (the highest, on Euler now)
  - The minor revision number indicates incremental changes within an architecture class

- A higher compute capability indicates an more able piece of hardware

- The “CUDA Version” indicates what version of the **software** you are using to run on the hardware
  - Right now, the most recent version of CUDA is 5.5

- In a perfect world
  - You would run the most recent CUDA (version 5.5) software release
  - You would use the most recent architecture (compute capability 3.0)
Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - Makes sense, the functionality in later versions increased, was not there in previous versions
NVIDIA CUDA Devices

- CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

<table>
<thead>
<tr>
<th>Card</th>
<th>Compute Capability</th>
<th>Number of Multiprocessors</th>
<th>Number of CUDA Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 690</td>
<td>3.0</td>
<td>2x8</td>
<td>2x1536</td>
</tr>
<tr>
<td>GTX 680</td>
<td>3.0</td>
<td>8</td>
<td>1536</td>
</tr>
<tr>
<td>GTX 670</td>
<td>2.1</td>
<td>7</td>
<td>1344</td>
</tr>
<tr>
<td>GTX 590</td>
<td>2.1</td>
<td>2x16</td>
<td>2x512</td>
</tr>
<tr>
<td>GTX 560TI</td>
<td>2.1</td>
<td>8</td>
<td>384</td>
</tr>
<tr>
<td>GTX 460</td>
<td>2.1</td>
<td>7</td>
<td>336</td>
</tr>
<tr>
<td>GTX 470M</td>
<td>2.1</td>
<td>6</td>
<td>288</td>
</tr>
<tr>
<td>GTX 450, GTX 460M</td>
<td>2.1</td>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>GT 445M</td>
<td>2.1</td>
<td>3</td>
<td>144</td>
</tr>
<tr>
<td>GT 435M, GT 425M, GT 420M</td>
<td>2.1</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>GT 415M</td>
<td>2.1</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>GTX 490</td>
<td>2.0</td>
<td>2x15</td>
<td>2x480</td>
</tr>
<tr>
<td>GTX 580</td>
<td>2.0</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>GTX 570, GTX 480</td>
<td>2.0</td>
<td>15</td>
<td>480</td>
</tr>
<tr>
<td>GTX 470</td>
<td>2.0</td>
<td>14</td>
<td>448</td>
</tr>
<tr>
<td>GTX 465, GTX 480M</td>
<td>2.0</td>
<td>11</td>
<td>352</td>
</tr>
<tr>
<td>GTX 295</td>
<td>1.3</td>
<td>2x30</td>
<td>2x240</td>
</tr>
<tr>
<td>GTX 285, GTX 280, GTX 275</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 260</td>
<td>1.3</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>9800 GX2</td>
<td>1.1</td>
<td>2x16</td>
<td>2x128</td>
</tr>
<tr>
<td>GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>512, GTX 285M, GTX 280M</td>
<td>8800 Ultra, 8800 GTX</td>
<td>1.0</td>
<td>16</td>
</tr>
<tr>
<td>9800 GT, 8800 GT</td>
<td>1.1</td>
<td>14</td>
<td>112</td>
</tr>
</tbody>
</table>
The CUDA Execution Model
GPU Computing – The Basic Idea

- The GPU is linked to the CPU by a reasonably fast connection

- The idea is to use the GPU as a co-processor
  - Farm out big parallel tasks to the GPU
  - Keep the CPU busy with the control of the execution and “corner” tasks
The CUDA Way: Extended C

- Declaration specifications:
  - global, device, shared, local, constant

- Keywords
  - threadIdx, blockIdx

- Intrinsics
  - __syncthreads

- Runtime API
  - Functions for memory and execution management

- Kernel launch

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx.x] = image[i];
  ...
  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMemcpy((void *)myimage, (void *)image, bytes, cudaMemcpyHostToDevice);

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
```
Computing on the GPU
CUDA and GPU Programming Model
Execution Configuration

September 25, 2013

“Computers are good at following instructions, but not at reading your mind.”
Donald Knuth
Before We Get Started…

- Last time
  - General discussion - computing on the GPU
  - Started the CUDA execution model

- Today
  - More on the CUDA execution model
  - The concept of “execution configuration”
  - Scheduling jobs on Euler
  - CMake : building the build process

- Miscellaneous
  - Third assignment posted and due on Monday at 11:59 PM
    - Has to do with GPU computing
  - Read pages 56 through 73 of the primer:
    [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
    - Please post suggestions for improvement
Good Source of Info


- Remark: Frustration often times stems from programming in C and not from the CUDA part
Example: Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Note the “cu” suffix

Output, on Euler:
```
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
Compiling with `nvcc` for CUDA

- Source files with CUDA language extensions must be compiled with `nvcc`
  - You spot such a file by its `.cu` or `.cuh` suffixes

- Example:
  ```
  $ nvcc -arch=sm_20 foo.cu
  ```

- `nvcc` is actually a compile driver
  - Works by invoking all the necessary tools and compilers like g++, cl, ...

- `nvcc` can output:
  - C code
    - Must then be compiled with the rest of the application using another tool
  - `ptx` code (CUDA’s assembly language, device independent)
  - Or directly object code (`cubin`)
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Two new syntactic elements…
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
  - People refer to it as being a “kernel”

- `nvcc` separates source code into host and device components
  - Device functions, e.g. `mykernel()`, processed by NVIDIA compiler
  - Host functions, e.g. `main()`, processed by standard host compiler
    - `gcc`, `cl.exe`
Hello World! with Device Code

mykernel<<<1,1>>>();

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - NOTE: we’ll return to the above (1,1) parameters soon

- That’s all that is required to execute a function on the GPU…
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output, on Euler:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

- Actually, `mykernel()` does not do anything yet...
Compiling CUDA Code

[with nvcc driver]

C/C++ CUDA Application

NVCC

PTX Code

PTX to Target Compile

K20X ...

C2050

Target binary code
PTX: Parallel Thread eXecution

- PTX: a pseudo-assembly language used in CUDA programming environment.
- `nvcc` translates code written in CUDA’s C into PTX
- `nvcc` subsequently invokes a compiler which translates the PTX into a binary code which can be run on a certain GPU

```c
__global__ void fillKernel(int *a, int n)
{
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    if (tid < n) {
        a[tid] = tid;
    }
}
```
### The nvcc Compiler – Suffix Info

<table>
<thead>
<tr>
<th>File suffix</th>
<th>How the nvcc compiler interprets the file</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cu</td>
<td>CUDA source file, containing host and device code</td>
</tr>
<tr>
<td>.cup</td>
<td>Preprocessed CUDA source file, containing host code and device functions</td>
</tr>
<tr>
<td>.c</td>
<td>'C' source file</td>
</tr>
<tr>
<td>.cc, .cxx, .cpp</td>
<td>C++ source file</td>
</tr>
<tr>
<td>.gpu</td>
<td>GPU intermediate file (device code only)</td>
</tr>
<tr>
<td>.ptx</td>
<td>PTX intermediate assembly file (device code only)</td>
</tr>
<tr>
<td>.cubin</td>
<td>CUDA device only binary file</td>
</tr>
</tbody>
</table>

The CUDA Execution Model is Asynchronous

This is how your C code looks like

This is how the code gets executed on the hardware in heterogeneous computing. GPU calls are asynchronous…
Languages Supported in CUDA

- Note that everything is done in C
  - Yet minor extensions are needed to flag the fact that a function actually represents a kernel, that there are functions that will only run on the device, etc.
    - You end up working in “C with extensions”

- FOTRAN is supported, we’ll not cover here though

- There is support for C++ programming (operator overload, new/delete, etc.)
  - Not fully supported yet
CUDA Function Declarations
(the “C with extensions” part)

<table>
<thead>
<tr>
<th>CUDA Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float myDeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void myKernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float myHostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function, launched by host, executed on the device
  - Must return void
- For a full list, see CUDA Reference Manual:
A kernel function must be called with an execution configuration:

```c
__global__ void kernelFoo(...); // declaration

dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block

kernelFoo<<< DimGrid, DimBlock >>>(...your arg list comes here...);
```

**NOTE**: Any call to a kernel function is asynchronous
- By default, execution on host doesn’t wait for kernel to finish.
Example

- The host call below instructs the GPU to execute the function (kernel) “foo” using 25,600 threads
  - Two arguments are passed down to each thread executing the kernel “foo”

```c
foo<<100,256>>(pMyMatrixD, pMyVecD)
```

- In this execution configuration, the host instructs the device that it is supposed to run 100 blocks each having 256 threads in it
- The concept of block is important since it represents the entity that gets executed by an SM (stream multiprocessor)
More on the Execution Model
[Some Constraints]

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of 65535 by 65535 by 65535 grid of blocks (about 280,000 billion blocks)

- Threads in each block:
  - The threads can be organized as a 3D structure (x,y,z)
  - The total number of threads in each block cannot be larger than 1024
Block and Thread Index (Idx)

- Threads and blocks have indices
  - Used by each thread to decide what data to work on (more later)
  - Block Index: a triplet of uint
  - Thread Index: a triplet of uint

- Why this 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on subdomains
    - …

 Courtesy: NVIDIA
A Couple of Built-In Variables
[Critical in supporting the SIMD parallel computing paradigm]

- It’s essential for each thread to be able to find out the grid and block dimensions and its block index and thread index.

- Each thread when executing a kernel has access to the following built-in variables:
  - `threadIdx` (uint3) – contains the thread index within a block
  - `blockDim` (dim3) – contains the dimension of the block
  - `blockIdx` (uint3) – contains the block index within the grid
  - `gridDim` (dim3) – contains the dimension of the grid
  - `[warpSize (uint) – provides warp size, we’ll talk about this later...]`
Thread Index vs. Thread ID

critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of “warp”

- Each block organizes its threads in a 3D structure defined by its three dimensions: $D_x$, $D_y$, and $D_z$ that you specify.

- A block cannot have more than 1024 threads $\Rightarrow D_x \times D_y \times D_z \leq 1024$.

- Each thread in a block can be identified by a unique index $(x, y, z)$, and

\[
0 \leq x < D_x \quad 0 \leq y < D_y \quad 0 \leq z < D_z
\]

- A triplet $(x, y, z)$, called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as $t_{id} = x + y \times D_x + z \times D_x \times D_y$. You can regard this as a ”projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on ”warps” later).

- In general, operating for vectors typically results in you choosing $D_y = D_z = 1$. Handling matrices typically goes well with $D_z = 1$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.
CUDA: This is it…

```c
#include <cutil_inline.h>
#include <iostream>

__global__ void simpleKernel(int* data)
{
    //write something trivial to the global memory...
    data[threadIdx.x] = blockIdx.x + threadIdx.x;
}

int main()
{
    int hostArray[4], *devArray;
    //allocate memory on the device (GPU)
    cudaMalloc((void**)&devArray, sizeof(int)*4);

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,4>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int)*4, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: ";
    std::cout << hostArray[0] << ", ";
    std::cout << hostArray[1] << ", ";
    std::cout << hostArray[2] << ", ";
    std::cout << hostArray[3] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);

    return 0;
}
```
Revisit - Execution Configuration: Grids and Blocks

- A kernel is executed as a **grid of blocks of threads**
  - All threads executing a kernel can access several device data memory spaces

- A **block [of threads]** is a collection of threads that can **cooperate** with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency **shared memory**

- **Exercise:**
  - How was the grid defined for this pic?
    - I.e., how many blocks in X and Y directions?
  - How was a block defined in this pic?
Example: Adding Two Matrices

- You have two matrices $A$ and $B$ of dimension $N \times N$ ($N=32$)
- You want to compute $C = A + B$ in parallel
- Code provided below (some details omitted, such as `#define N 32`

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
            float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of $N \times N \times 1$ threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Something to think about…

- Given that the x field of a thread index changes the fastest, is the array indexing scheme on the previous slide good or bad?

- The “good or bad” refers to how data is accessed in the device’s global memory.

- In other words should we have

\[
C_{ij} = A_{ij} + B_{ij}
\]

or...

\[
C_{ji} = A_{ji} + B_{ji}
\]
There are three rules to follow when parallelizing large codes.
Unfortunately, no one knows what these rules are.”
W. Somerset Maugham and Gary Montry
Before We Get Started…

- **Last time**
  - Covered the “execution configuration”
  - Discussion, thread index vs. thread ID

- **Today**
  - Example, working w/ large arrays
  - Timing a kernel execution
  - The CUDA API
  - The memory ecosystem

- **Miscellaneous**
  - Third assignment posted and due on Monday at 11:59 PM
    - Has to do with GPU computing
  - Read pages 56 through 73 of the primer: [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
    - Please post suggestions for improvement
Example: Array Indexing

- Purpose of Example: see a scenario of how multiple blocks are used to index entries in an array

- First, recall this: there is a limit on the number of threads you can squeeze in a block (up to 1024 of them)

- **Note**: In the vast majority of applications you need to use many blocks (each of which contains the same number N of threads) to get a job done. This example puts things in perspective
Example: Array Indexing
[Important to grasp: shows thread to task mapping]

- No longer as simple as using only `threadIdx.x`
  - Consider indexing into an array, one thread accessing one element
  - Assume you have \( M=8 \) threads per block and the array is 32 entries long

\[
\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;
\]

[NVIDIA]→
Example: Array Indexing

What will be the array entry that thread of index 5 in block of index 2 will work on?

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```

```
Imagine you are one of many threads, and you have your thread index and block index

- You need to figure out what the work you need to do is
  - Just like we did on previous slide, where thread 5 in block 2 mapped into 21

- You have to make sure you actually need to do that work
  - In many cases there are threads, typically of large id, that need to do no work
  - Example: you launch two blocks with 512 threads but your array is only 1000 elements long. Then 24 threads at the end do nothing
Before Moving On…
[Some Words of Wisdom]

- In GPU computing you launch as many threads as data items (tasks, jobs) you have to perform
  - This replaces the purpose in life of the “for” loop
  - Number of threads & blocks is established at run-time

- Number of threads = Number of data items (tasks)
  - It means that you’ll have to come up with a rule to match a thread to a data item (task) that this thread needs to process
  - Solid source of errors and frustration in GPU computing
    - It never fails to deliver (frustration)
      :-(

:-(
Timing Your Application

Timing support – part of the CUDA API
- You pick it up as soon as you include `<cuda.h>`

Why it is good to use
- Provides cross-platform compatibility
- Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

Reports time in milliseconds, accurate within 0.5 microseconds
- From NVIDIA CUDA Library Documentation:
  - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`. If either event has been recorded with a non-zero stream, the result is undefined.
Timing Example

~ Timing a query of device 0 properties ~

```cpp
#include <iostream>
#include <cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);
    cudaEventRecord(startEvent, 0);

    cudaDeviceProp deviceProp;
    const int currentDevice = 0;
    if (cudaGetDeviceProperties(&deviceProp, currentDevice) == cudaSuccess)
        printf("Device %d: %s\n", currentDevice, deviceProp.name);
    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";
    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```
The CUDA API
What is an API?

- Application Programming Interface (API)
  - “A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs” (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This is what replaced old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program
On the CUDA API

- Reading the CUDA Programming Guide you’ll run into numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs `nvcc` compiler to generate an executable

- CUDA Driver API – low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly clunkier way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
CUDA runtime API: exposes a set of extensions to the C language

- Spelled out in an appendix of “NVIDIA CUDA C Programming Guide”
- There is many of them → Keep in mind the 20/80 rule

CUDA runtime API:

- **Language extensions**
  - To target portions of the code for execution on the device

- A runtime library, which is split into:
  - A common component providing built-in vector types and a subset of the C runtime library available in both host and device codes
    - Callable both from device and host
  - A host component to control and access devices from the host
    - Callable from the host only
  - A device component providing device-specific functions
    - Callable from the device only
### Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong></td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- **__device__** is optional when used with **__local__**, **__shared__**, or **__constant__**.

- **Automatic variables** without any qualifier reside in a **register**
  - **Except arrays**, which reside in local memory (unless they are small and of known constant size).
Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host.

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes
**Common Runtime Component:**

**Built-in Vector Types**

- \([u]\text{char}[1..4], [u]\text{short}[1..4], [u]\text{int}[1..4], [u]\text{long}[1..4], \text{float}[1..4], \text{double}[1..2]\)
- Structures accessed with \(x, y, z, w\) fields:
  
  ```c
  uint4 param;
  int dummy = param.y;
  ```

- \(\text{dim3}\)
  - Based on \(\text{uint3}\)
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in "NVIDIA CUDA C Programming Guide"
Common Runtime Component: Mathematical Functions

- pow, sqrt, cbrt, hypot
- exp, exp2, expm1
- log, log2, log10, log1p
- sin, cos, tan, asin,acos, atan, atan2
- sinh, cosh, tanh, asinh, acosh, atanh
- ceil, floor, trunc, round
- etc.

- When executed on the host, a given function uses the C runtime implementation if available
- These functions only supported for scalar types, not vector types
Host Runtime Component

- Provides functions available only to the host to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Examples
  - Device memory allocation
    - `cudaMalloc()`, `cudaFree()`
  - Memory copy from host to device, device to host, device to device
    - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
  - Memory addressing – returns the address of a device variable
    - `cudaGetSymbolAddress()`
CUDA API: Device Memory Allocation

[Note: picture assumes two blocks, each with two threads]

- `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- `cudaFree()`
  - Frees object from device Global Memory
  - Pointer to freed object
Example Use: A Matrix Data Type

- NOT part of CUDA API
- Used in several code examples
  - 2 D matrix
  - Single precision float elements
  - width * height entries
  - Matrix entries attached to the pointer-to-float member called "elements"
  - Matrix is stored row-wise

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Code example:

- Allocate a 64 * 64 single precision float array
- Attach the allocated storage to Md.elements
- “d” in “Md” is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**) &Md.elements, size);
...
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

**Question:** why is the type of the first argument (void **)?
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 x 64 single precision float array
  - M is in host memory and Md is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. $\_\_\_\sin(x)$)
  - $\_\_\_\text{pow}$
  - $\_\_\_\text{log}$, $\_\_\_\text{log2}$, $\_\_\_\text{log10}$
  - $\_\_\_\text{exp}$
  - $\_\_\_\sin$, $\_\_\_\cos$, $\_\_\_\tan$

- Some of these have hardware implementations

- By using the “-use\_fast\_math” flag, $\sin(x)$ is substituted at compile time by $\_\_\_\sin(x)$

```bash
>> nvcc -arch=sm_20 -use\_fast\_math foo.cu
```
CPU vs. GPU – Flop Rate (GFlops)
Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Use only global memory (don’t bring shared memory into picture yet)
  - Matrix will be of small dimension, job can be done using one block
  - Concentrate on
    - Thread ID usage
  - Memory data transfer API between host and device
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $W \times W$
  - Assume $W$ was defined to be 32

- Software Design Decisions:
  - One **thread** handles one element of $P$
  - Each thread will access all the entries in one row of $M$ and one column of $N$
    - $2W$ read accesses to global memory
    - One write access to global memory
Multiply Using One Thread Block

- One Block of threads computes matrix $P$
  - Each thread computes one element of $P$

- Each thread
  - Loads a row of matrix $M$
  - Loads a column of matrix $N$
  - Perform one multiply and addition for each pair of $M$ and $N$ elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…

- Size of matrix limited by the number of threads allowed in a thread block
Matrix Multiplication: Traditional Approach, Coded in C

// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P) {
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; //march along a row of M
                double b = N.elements[k * N.width + j]; //march along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
Step 2: Matrix Multiplication
[host-side code]

```c
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P) {
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```
Matrix multiplication kernel – thread specification

```c
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
  // 2D Thread Index; computing P[ty][tx]...
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue will end up storing the value of P[ty][tx].
  // That is, P.elements[ty * P. width + tx] = Pvalue
  float Pvalue = 0;

  for (int k = 0; k < M.width; ++k) {
    float Melement = M.elements[ty * M.width + k];
    float Nelement = N.elements[k * N.width + tx];
    Pvalue += Melement * Nelement;
  }

  // Write matrix to device memory; each thread one element
  P.elements[ty * P. width + tx] = Pvalue;
}
```

Step 4: Matrix Multiplication- Device-side Kernel Function
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMemcpy((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.

Edsger Dijkstra
Before We Get Started…

- Last time
  - Example, working w/ large arrays
  - Timing a kernel execution
  - The CUDA API

- Today
  - The memory ecosystem

- Miscellaneous
  - Fourth assignment will be posted today and due on Monday, October 7, at 11:59 PM
    - GPU computing related
    - Kicks off a series of four challenging assignments
  - Read pages 73 through its end: http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWInterface.pdf
    - Please post suggestions for improvement
Before Diving In: A Word On CUDA Streams

- A sequence of CUDA calls should be visualized as belonging to a stream.

- IMPORTANT: All CUDA calls (items) in a stream are strictly executed in the order in which they were “deposited” into this stream AND no item in the stream starts before the following item in the stream finishes.
  - Example:
    - Imagine you have a sequence of cudaMemcpy-1 followed by kernel-call followed by cudaMemcpy-2.
    - cudaMemcpy-2 will not start before the GPU finishes execution of the kernel-call.

- Recall that kernel calls are asynchronous (implications when timing calls).

- cudaMemcpy() is synchronous (blocks the execution of CPU).
  - There is an asynchronous version as:
    - Even for the asynchronous version, the strict execution order in the stream is observed.
End API discussion
…… transitioning into...
The Memory Ecosystem
Fermi: Global Memory

- Up to 6 GB of “global memory”
- “Global” in the sense that it doesn’t belong to an SM but rather all SM can access it
GPU vs. CPU – Memory Bandwidth

Theoretical GB/s

- CPU
- GeForce GPU
- Tesla GPU

- GeForce GTX TITAN
- Tesla K20X
- Tesla M2090
- GeForce GTX 680
- Tesla C2050
- GeForce GTX 480
- Tesla C1060
- GeForce 8800 GTX
- GeForce 7800 GTX
- Sandy Bridge
- GeForce 6800 GT
- GeForce FX 5900
- Northwood
- Prescott
- Woodcrest
- Harpertown
- Bloomfield
- Sandy Bridge
- Westmere
The Fermi Architecture

- 64 KB L1 cache & shared memory
- 768 KB L2 uniform cache (shared by all SMs)
- Memory operates at its own clock rate
- High memory bandwidth
  - Close to 200 GB/s
CUDA Device Memory Space Overview
[Note: picture assumes two blocks, each with two threads]

- Image shows the memory hierarchy that a block sees while running on an SM

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory

**IMPORTANT NOTE:** Global, constant, and texture memory spaces are **persistent** across kernels called by the same host application.
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

NOTE: We will not emphasize texture memory in this class.
The Concept of Local Memory

- Note the presence of local memory, which is virtual memory
  - If too many registers are needed for computation ("high register pressure") the ensuing data overflow is stored in local memory
  - "Local" means that it’s local, or specific, to one thread
  - In fact local memory is part of the global memory
  - Long access times for local memory (in Fermi, local memory is cached)
## Storage Locations

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A – resident</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A – resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Off-chip means on-device; i.e., slow access time.
Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
The Three Most Important Parallel Memory Spaces

- **Register**: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit if not cached)
- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- **Global Memory**: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication
SM Register File (RF) [Tesla C1060]

- Register File (RF)
  - 64 KB (Tesla: 16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core

- TEX pipe can also read/write RF

- Global Memory Load/Store pipe can also read/write RF
**Programmer View of Register File**

- Number of **32 bit** registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla
  - 32K on Fermi
  - 64K on Kepler

- Size of Register File dependent on your compute capability

- Registers are *dynamically partitioned* across all Blocks assigned to the SM

- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks

- A thread in a Block can only access registers assigned to itself
  - Kepler: a thread can have up to 255 registers

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example

[Tesla C1060]

- If each Block has 16X16 threads and each thread uses 20 registers, how many blocks can run on each SM?
  - Each Block requires 20*256 = 5120 registers
  - 16,384 = 3 * 5120 + pocket change
  - As such, three blocks can run on an SM as far as registers are concerned

- What if each thread increases the use of registers from 20 to 22?
  - Each Block now requires 22*256 = 5632 registers
  - 16,384 < 16896 = 5632 *3
  - Only two Blocks can run on an SM, about 33% reduction of parallelism!!!

- This example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism
  - One way to find out how many registers you use per thread is to invoke the compile flag `-ptx-options=-v` when you compile with `nvcc`
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.

- Tradeoff between instruction-level parallelism (CPU) and thread level parallelism (GPU)
  - TLP: many threads are launched
  - ILP: few threads are launched, but for each thread several instructions can be executed simultaneously
Constant Memory

- This comes handy when all threads use the same *constant* value in their computation
  - Example: $\pi$, some spring force constant, $e=2.7173$, etc.

- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)

- A constant value can be broadcast to all threads in a warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block
  - When all threads in a warp read the same constant memory address this is as fast as a register
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of “printf”]

```c
#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;

    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>();  // Some dummy function that uses the constant variable
    cudaThreadSynchronize();

    return 0;
}
```

Hello thread 0, Circ= 0.00
Hello thread 1, Circ= 6.28
Hello thread 2, Circ=12.57
Hello thread 3, Circ=18.85
Hello thread 4, Circ=25.13
Hello thread 5, Circ=31.42
Hello thread 6, Circ=37.70
Hello thread 7, Circ=43.98
Hello thread 8, Circ=50.27
Hello thread 9, Circ=56.55
Hello thread 10, Circ=62.83
Hello thread 11, Circ=69.11
Hello thread 12, Circ=75.40
Hello thread 13, Circ=81.68
Hello thread 14, Circ=87.96
Hello thread 15, Circ=94.25
Matrix Multiplication Example, Revisited

- **Purpose**
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the `_syncthreads()` function call

- **NOTE:** A one dimensional array stores the entries in the matrix
Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction ("arithmetic intensity") very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device N.width times
  - The matrix N (its entries) is copied from global memory to the device M.height times

- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things
The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU.

- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined.

- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up…). You’re done.

**Rules of Thumb for Efficient GPU Computing:**
1. Get the data on the GPU and keep it there.
2. Give the GPU enough work to do.
3. Focus on data reuse within the GPU to avoid memory bandwidth limitations.
A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition (“tile”) data to take advantage of fast shared memory:
  - Partition data into data subsets (tiles) that each fits into shared memory
  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory back to global memory
Multiply Using Several Blocks

- One block computes one square sub-matrix $C_{\text{sub}}$ of size `Block_Size`
- One thread computes one entry of $C_{\text{sub}}$
- Assumption: $A$ and $B$ are square matrices and their dimensions of are multiples of `Block_Size`
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with `Block_Size=16x16`

**NOTE 1:** Similar example provided in the CUDA Programming Guide 3.2
- Available on the 2011 class website

**NOTE 2:** A similar technique is used on CPUs to improve cache hits. See slide “Blocking Example” at
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

The Memory Ecosystems, Wrap-up
Scheduling Issues in CUDA

October 2, 2013

Success consists of going from failure to failure without loss of enthusiasm.
Winston Churchill
Before We Get Started...

- Last time
  - The memory ecosystem

- Today
  - Example: tiled matrix multiplication → introduces the concept of Shared Memory
  - Execution scheduling

- Miscellaneous
  - Fourth assignment due on Monday, October 7, at 11:59 PM
    - GPU computing related
    - Kicks off a series of four challenging assignments
  - Read pages 73 through its end: [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
    - Please post suggestions for improvement
  - Half way through the semester: I’m asking for your feedback
    - To be provided anonymous on Monday, Oct 7 – details to follow in an email on Friday
    - I’ll compile all of them and upload on the class website
Matrix Multiplication Example, Revisited

- **Purpose**
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the \texttt{__syncthreads()} function call

- **NOTE:** A one dimensional array stores the entries in the matrix
Why Revisit the Matrix Multiplication Example?

- In the naïve first implementation the ratio of arithmetic computation to memory transaction (“arithmetic intensity”) very low
  - Each arithmetic computation required one fetch from global memory
  - The matrix M (its entries) is copied from global memory to the device N.width times
  - The matrix N (its entries) is copied from global memory to the device M.height times

- When solving a numerical problem the goal is to go through the chain of computations as fast as possible
  - You don’t get brownie points moving data around but only computing things
The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU

- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined

- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up...). You’re done.

**Rules of Thumb for Efficient GPU Computing:**
1. Get the data on the GPU and keep it there
2. Give the GPU enough work to do
3. Focus on data reuse within the GPU to avoid memory bandwidth limitations
Local and global memory reside in device memory (DRAM) - much slower access than shared memory

An advantageous way of performing computation on the device is to partition (“tile”) data to take advantage of fast shared memory:

- **Partition** data into data subsets (tiles) that each fits into shared memory

- **Handle** each data subset (tile) with one thread block by:
  - Loading the tile from global memory into shared memory, using multiple threads to exploit memory-level parallelism
  - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
Multiply Using Several Blocks

- One block computes one square sub-matrix $C_{\text{sub}}$ of size $\text{Block}_\text{Size}$

- One thread computes one entry of $C_{\text{sub}}$

- **Assumption:** $A$ and $B$ are *square matrices* and their dimensions are *multiples* of $\text{Block}_\text{Size}$
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
  - In this example work with $\text{Block}_\text{Size}=16x16$

**NOTE 1:** Similar example provided in the CUDA Programming Guide 3.2
- Available on the 2011 class website

**NOTE 2:** A similar technique is used on CPUs to improve cache hits. See slide “Blocking Example” at [http://cseweb.ucsd.edu/classes/fa10/cse240a/pdf/08/CSE240A-MBT-L15-Cache.ppt.pdf](http://cseweb.ucsd.edu/classes/fa10/cse240a/pdf/08/CSE240A-MBT-L15-Cache.ppt.pdf)
A Block of 16 X 16 Threads

(tx=0, ty=0) (1,0) (2,0) (15,0)
(0,1) (16,1) (15,1)
(0,15) (1,15) (15,15)
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C) {
    int size;

    // Load A and B to the device
    float* Ad; size = hA * wA * sizeof(float); cudaMalloc((void**)&Ad, size); cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd; size = wA * wB * sizeof(float); cudaMalloc((void**)&Bd, size); cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd; size = hA * wB * sizeof(float); cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE); dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
First entry of the tile

(number of tiles along the width of B)

(number of tiles down the height of A)

aBegin

aStep

bBegin

bStep
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Shared memory for the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Shared memory for the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
Synchronization Function

- It's a device lightweight runtime API function
  - `void __syncthreads();`

- Synchronizes all threads **in a block** (acts as a barrier for all threads of a block)
  - Does **not** synchronize threads from two blocks

- Once all threads have reached this point, execution resumes normally

- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory

- Allowed in conditional constructs only if the conditional is uniform across the entire thread block
The Cache vs. Shared Mem. Conundrum

- On Fermi and Kepler you can split some fast memory between shared memory and cache
  - Fermi: you can go 16/48 or 48/16 KB for ShMem/Cache
  - Lots of Cache & Little ShMem:
    - Handled for you by the scheduler
    - No control over it
    - Can’t have too many blocks of threads running if blocks use ShMem
  - Lots of ShMem & Little Cache:
    - Good in tiling, if you want to have full control
    - ShMem pretty cumbersome to manage
Memory Issues Not Addressed Yet…

- Not all global memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question

- Not all shared memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions

- To do justice to these topics we’ll need to talk first about scheduling threads for execution
  - Next course segment…
Why Do We Do This?

- Hone our “Computational Thinking” skills

- “Computational Thinking” cannot be built without
  - Working on our programming skills
    and more importantly,
  - Gaining a good understanding of how the hardware supports the execution of your code (the hardware/software interplay)

- Good programming skills ensures we get correct results
- Computational thinking allows we to get the correct results fast
Execution Scheduling Issues
[NVIDIA cards specific]
Thread Execution Scheduling

- **Topic we are about to discuss:**
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM. How is this possible?
CUDA Thread Block

[We already know this…]

- In relation to a Block, the programmer decides:
  - Block size: from 1 to 1024 threads
  - Block dimension (shape): 1D, 2D, or 3D
  - Higher order configurations projected to 1D representation

- Threads have thread idx numbers within Block
- Threads within Block share data and may synchronize while each is doing its work
- Thread program uses thread idx to select work and address shared data
- Beyond the concept of thread idx we brought into the picture the concept of thread id and how to compute a thread id based on the thread index (the 1D projection idea)
The 30,000 Feet Perspective

- There are two schedulers at work in GPU computing
  - A device-level scheduler: assigns blocks to SM that indicate at a given time “excess capacity”
  - An SM-level scheduler, which schedules the execution of the threads in a block onto the functional units available to an SM
  - The more interesting is the SM-level scheduler
Device-Level Scheduler

- Grid is launched on the device
- Thread Blocks are serially distributed to all the SMs
  - Potentially more than one block per SM
- As Thread Blocks complete kernel execution, resources are freed
  - Device-level scheduler can launch next Block[s] in line
- This is the first levels of scheduling:
  - For running [desirably] a large number of blocks on a relatively small number of SMs (16/14/etc.)
- Limits for resident blocks:
  - 16 blocks can be resident on a Kepler SM
  - 8 blocks can be resident on a Fermi & Tesla SM
SM-Level Scheduler[s]

- Each Thread Block divided in 32-thread “Warps”
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling unit in SM

- Limits, number of resident warps on an SM:
  - 64 warps on Kepler (i.e., 2048 resident threads)
  - 48 warps on Fermi (i.e., 1536 resident threads)
  - 32 warps on Tesla (i.e., 1024 resident threads)

- EXAMPLE: If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- Cycles needed to dispatch the same instruction for all threads in a warp
  - On Tesla: 4 cycles
  - On Fermi: 1 cycle

- How is this relevant?
  - Suppose you use a Tesla card AND our code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

\[
\frac{400}{(6 \times 4)} = 16.6667 \Rightarrow 17 \text{ Warps}
\]
SM Instruction Buffer – Warp Scheduling [Tesla gen]

- Fetch one warp instruction/cycle
  - From instruction L1 cache
  - Into any instruction buffer slot

- Issue one “ready-to-go” warp instruction per 4 cycles
  - From any warp - instruction buffer slot
  - Operand scoreboard used to prevent hazards

- Issue selection based on round-robin/age of warp

- SM broadcasts the same instruction to 32 Threads of a Warp
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads.
- One warp issued at each clock cycle by each scheduler.
- During no cycle can more than 2 warps be dispatched for execution on the four functional units.
- Scoreboarding is used to figure out which warp is ready.
Scoreboarding

- Used to determine whether a warp is ready to execute

- A **scoreboard** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

- Old concept from CDC 6600 (1960s) to separate memory and computation
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Not eligible</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Not eligible</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
</tbody>
</table>

Mary Hall, U-Utah
Scoreboarding from Example

Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Ready to write result</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Not eligible</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Not eligible</td>
</tr>
</tbody>
</table>

Schedule at time k+1

Mary Hall, U-Utah
Example: Fermi Related

- Scheduler works at 607 MHz
- Functional units work at 1215 MHz

Question:
- What is the peak flop rate of GTX480?
  - 15 SMs * 32 SPs * 1215 * 2 (Fused Multiplied Add) = 1166400 Mflopz
  - That is, 1.166 Tflops, single precision
Fermi Specifics

- As illustrated in the picture, at no time can we see more than 2 warps being dispatched for execution during a cycle.
- Note that at any given time we might have more than two functional units working though (which is actually very good, we keep it busy).
Scheduling Issues in CUDA
Global Memory Access Patterns

October 4, 2013
Before We Get Started…

- Last time
  - Example: tiled matrix multiplication → introduced the concept of Shared Memory
  - Execution scheduling

- Today
  - Wrap up discussion on execution scheduling
  - Discuss global memory access patterns

- Miscellaneous
  - Fourth assignment due on Monday, October 7, at 11:59 PM
    - GPU computing related
    - Kicks off a series of four challenging assignments
  - Read pages 73 through its end: [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
    - Please post suggestions for improvement
  - Half way through the semester: I’m asking for your feedback
    - To be provided anonymous on Wednesday, Oct 9 – details to follow in an email on Monday
    - I’ll compile all of them and upload on the class website
  - Syllabus updated on the course website
Technical Specifications and Features

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
“multiprocessor” stands for Stream Multiprocessor (what we called SM)

This is us: most GPUs on Euler are Fermi

<table>
<thead>
<tr>
<th>Feature Support</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory (Section B.11)</td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Section B.12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>__ballot() (Section B.12)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__threadfence_system() (Section B.5)</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>
Threads are Executed in Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently
Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing
  - Warp scheduling is not something you control in CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion
Revisiting the Concept of Execution Configuration

- Prefer thread block sizes that result in mostly full warps

**Bad:** kernel<<<N, 1>>> ( ... )
**Okay:** kernel<<<(N+31) / 32, 32>>> ( ... )
**Better:** kernel<<<(N+127) / 128, 128>>> ( ... )

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency

- Resource like __shared__ may constrain number of threads per block

- Algorithm and decomposition of problem will reveal the preferred amount of shared data and __shared__ allocation
  - We often have to take a step back and come up with a new algorithm that exposes parallelism

NVIDIA [J. Balfour]→
Scheduling: Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.

- Up to 8 blocks (on Fermi) can be executed at the same time by an SM.

- When a block of threads is executed on an SM, its threads are grouped in warps. The SM executes several warps at the same time.

- When a thread block finishes, a new block is launched on the vacated SM.
Granularity Considerations

[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory) of last lecture, should I use 8X8, 16X16 or 64X64 threads per blocks?
  - For 8X8, we have 64 threads per Block. Since each Fermi SM can manage up to 1536 resident threads, it could take up to 24 Blocks. However, each SM is limited to 8 resident Blocks, so only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each Fermi SM can take up to 1536 resident threads, it can take up to 6 Blocks unless other resource considerations overrule.
    - Next you need to see how much shared memory and how many registers get used in order to understand whether you can actually have four blocks per SM
  - 64X64 is a no starter, you can only have up to 1024 threads in a block, the tile cannot be this big

- NOTE: this is the “computational thinking” we discussed last time
Example: More Warps Not Always Better

[C1060 specific]

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers.

- Also, assume global loads have an associated overhead of 400 cycles.
  - 3 Blocks can run on each SM; i.e., 24 warps.

- If a compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load:
  - Only two blocks can now run on each SM.
  - However, one only needs 400 cycles/(8 instructions *4 cycles/instruction) \( \approx 13 \) Warps to tolerate the memory latency.
  - Two Blocks have 16 Warps. The performance can be actually higher!
A Word on HTT

The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency.

Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations.

For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.

HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.

This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.

Similar to the use of multiple warps on the GPU to hide latency

The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060).
Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors in response to AMD's 3DNow!

- SSE contains 70 new instructions

Example

- Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

\[
\begin{align*}
\text{vec}\_\text{res}.x & = v1.x + v2.x; \\
\text{vec}\_\text{res}.y & = v1.y + v2.y; \\
\text{vec}\_\text{res}.z & = v1.z + v2.z; \\
\text{vec}\_\text{res}.w & = v1.w + v2.w;
\end{align*}
\]

- SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

\[
\begin{align*}
\text{movaps xmm0, address-of-v1} &: \text{ xmm0=}v1.w | v1.y | v1.x \\
\text{addps xmm0, address-of-v2} &: \text{ xmm0=}v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x \text{ movaps address-of-vec\_res,xmm0}
\end{align*}
\]
Thread Divergence

Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

Half the threads in the warp execute the `if` clause, the other half the `else` clause.
Thread Divergence

[2/4]

- The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

- Often, this requires that the hardware execute multiple paths through a kernel for a warp.
  - For example, both the `if` clause and the corresponding `else` clause.
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
Thread Divergence

[4/4]

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one does need to consider divergence when reasoning about the performance of a program

- NVIDIA calls execution model SIMT (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here’s an extreme example...

```c
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
    case 0 : x[i] = a(x[i]);
             break;
    case 1 : x[i] = b(x[i]);
             break;
    ...
    case 31: x[i] = v(x[i]);
             break;
    }
}
```
Performance of Divergent Code

- Compiler and hardware can detect when all threads in a warp branch in the same direction
  - Example: all take the `if` clause, or all take the `else` clause
  - The hardware is optimized to handle these cases without loss of performance
  - In other words, use of `if` or `switch` does not automatically translate into disaster:

  ```
  if (threadIdx.x / WARP_SIZE >= 2) { }
  ```
  - Creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

- The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses
Global Memory Access Issues
Global Memory and Memory Bandwidth

- Memory attributes change from card to card. On Tesla C1060:
  - 4 GB in GDDR3 RAM
  - Memory clock speed: 800 MHz
  - Memory interface: 512 bits
  - Peak Bandwidth: \(800 \times 10^6 \times (512/8) \times 2 = 102.4 \text{ GB/s}\)

- When reporting effective bandwidth of your application:
  - Formula, effective bandwidth \((B_r - \text{bytes read}, B_w - \text{bytes written})\) [measured in GB/s]
    \[\text{Effective bandwidth} = \frac{(B_r + B_w) \times 10^9}{\text{time}}\]
  - Example: kernel copies a \(2048 \times 2048\) matrix from global memory, then copies matrix back to global memory. Does it in a certain amount of \(\text{time}\) [measured in seconds]
    \[\text{Effective bandwidth} = \frac{(2048^2 \times 4 \times 2) \times 10^9}{\text{time}}\]
  - 4 above comes from four bytes per float, 2 from the fact that the matrix is both read from and written to the global memory. The \(10^9\) used to get an answer in GB/s.
Data Access “Divergence”

- Concept is similar to thread divergence and often conflated

- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores

- If a warp doesn’t access a contiguous block of global memory the effective bandwidth is getting reduced

- Remember this: when you look at a kernel you see what a collection of threads; i.e., a warp, is supposed to do in lockstep fashion
Global Memory

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The layout of the access to global memory (the pattern of the access)
  - The size/alignment of the data you try to fetch from global memory
“Memory Access Layout”
What is it?

- The basic idea:
  - Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel.
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place.
  - Case (a) is called a “coalesced memory access”:
    - If you end up with (b) this will adversely impact the overall program performance.

- Analogy:
  - Can send one truck on six different trips to bring back each time a bundle of wood.
  - Alternatively, can send truck to one place and get it back fully loaded with wood.
Memory Facts, Fermi GPUs

- There is 64 KB of fast memory on each SM that gets split between L1 cache and Shared Memory
  - You can split 64 KB as “L1/Sh: 16/48” or “L1/Sh: 48/16”

- L2 cache: 768 KB – one big pot available to *all* SMs on the device

- L1 and L2 cache used to cache accesses to
  - Local memory, including register spill
  - Global memory

- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction
Fermi Memory Layout

[credits: NVIDIA]
## GPU-CPU Face Off

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050 (Fermi)</th>
<th>CPU – Intel core I7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Cores</td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td>Memory</td>
<td>64* KB L1, per SM 768 KB L2, all SMs 3 GB Device Mem.</td>
<td>- 32 KB L1 cache / core - 256 KB L2 (I&amp;D)cache / core - 8 MB L3 (I&amp;D) shared, all cores</td>
</tr>
<tr>
<td>Clock speed</td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>Floating point operations/s</td>
<td>$515 \times 10^9$ Double Precision</td>
<td>$70 \times 10^9$ Double Precision</td>
</tr>
</tbody>
</table>

* - split 48/16
More Memory Facts
[Fermi GPUs]

- All global memory accesses are cached

- A cache line is 128 bytes
  - It maps to a 128-byte aligned segment in device memory
  - Note: it so happens that 128 bytes = 32 (warp size) * 4 bytes
    - In other words, 32 floats or 32 ints can be brought over in fell swoop

- You can determine at *compile* time (through flags: `-dlcm=ca/cg`) if you double cache [L1 & L2] or only cache [L2 only]
  - If [L1 & L2], a memory access is serviced with a 128-byte memory transaction
  - If [L2 only], a memory access is serviced with a 32-byte memory transaction
    - This can reduce over-fetch in the case of scattered memory accesses
    - Good for irregular pattern access (sparse linear algebra)
More Memory Facts
[Fermi GPUs]

- If the size of the type accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently.

- The memory access schema is as follows:
  - Two memory requests, one for each half-warp, if the size is 8 bytes.
  - Four memory requests, one for each quarter-warp, if the size is 16 bytes.

- Each memory request is then broken down into cache line requests that are issued independently.

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.
More Memory Facts
[Fermi GPUs]

- When it comes to memory store transactions to global memory:
  - First, the L1 cache is invalidated if need be
  - Next, the data is stored in L2
  - The data is actually written to global memory only if/when the data gets evicted from L2

- This strategy works since L2 is visible to all SMs on the device (unlike L1)

- How about read-before-write issues?
  - Use atomic operations (discussed next lecture)
How to Use L1 and L2

- Should you start programming to leverage L1 and L2 cache?
  - The answer is: NO
    - GPU caches are not intended for the same use as CPU caches
      - Smaller sizes (on a per-thread basis, that is), not aimed at temporal reuse
        - Intended to smooth out some access patterns, help with spilled registers, etc.
    - Don’t try to block for L1/L2 like you would on CPU
      - You have 100s to 1000s of run-time scheduled thread hitting the caches
      - Instead of L1, you should start thinking how to leverage Shared Memory
        - Same bandwidth (they *physically* share the same memory banks)
        - Hardware will not evict behind your back

- Conclusions
  1. Optimize as if no caches were there
  2. The reason why we talk about this: it helps you understand when the GPU is good and when it’s not
Examples of Global Mem. Access by a Warp

- **Setup:**
  - You want to access floats or integers
  - In other words, each thread is requesting a 4-Byte word

- **Scenario A: access is aligned and sequential**

  ![Diagram showing aligned and sequential access]

  - **Good to know:** any address of memory allocated with `cudaMalloc` is a multiple of 256
  - That is, the addressed is 256 byte aligned, which is stronger than 128 byte aligned
Examples of Global Mem. Access by a Warp

[Cntd.]

- Scenario B: Aligned but non-sequential

- Scenario C: Misaligned and sequential
Why is this important?

- Compare Scenario B to Scenario C

- Basically, you have in Scenario C half the effective bandwidth you get in Scenario B
  - Just because of the alignment of your data access

- If your code is memory bound and dominated by this type of access, you might see a doubling of the run time…

- The moral of the story:
  - When you reach out to fetch data from global memory, visualize how a full warp reaches out for access. Is the access coalesced and well aligned?

- Scenarios A and B: illustrate what is called a coalesced memory access
Test Your Understanding

- Say you use in your program complex data constructs that could be organized using C-structures.

- Based on what we’ve discussed so far today, how is it more advantageous to store data in global memory?
  - Alternative A: as an array of structures
  - Alternative B: as a structure of arrays
Example: Adding Two Matrices

- You have two matrices A and B of dimension $N \times N$ ($N=32$)
- You want to compute $C = A + B$ in parallel
- Code provided below (some details omitted, such as `#define N 32`)

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                        float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N \times N \times 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Given that the x field of a thread index changes the fastest, is the array indexing scheme on the previous slide good or bad?

The “good or bad” refers to how data is accessed in the device’s global memory.

In other words should we have

\[ C[i][j] = A[i][j] + B[i][j] \]

or...

\[ C[j][i] = A[j][i] + B[j][i] \]
ME759
High Performance Computing for Engineering Applications

Shared Memory
Synchronization for Communication
 Atomic Operations
CUDA Optimization

October 7, 2013
Before We Get Started…

- Last time
  - Execution scheduling issues
  - Discussion of global memory access patterns

- Today
  - Shared memory, further considerations
  - Synchronization for Data Communication under CUDA
  - Atomic operations
  - CUDA Optimization/Best Practices issues

- Miscellaneous
  - Fourth assignment due tonight at 11:59 PM
  - Fifth assignment posted later today. GPU computing related
  - We’re half way through this class: please let me know what you think
    - Please provide feedback on Wednesday, Oct 9 – details to follow in an email
    - I’ll compile all of your feedback and upload on the class website
  - Exam: Th, November 7, 7-9 PM (no class on Friday, Nov. 8)
    - Review session on Wd, Nov. 6 @ 6 PM
    - Exam will draw on material covered in class and information provided in the primer
    - It’ll be a pen and paper exam. Open book and open anything
Shared Memory: Syntax & Semantics

- You can statically declare shared memory like in the code snippet below:

```c
__global__ void coalescedMultiply(float *a, float* b, float *c, int N) {
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

- The variable `aTile` visible to all threads in each block, and only to those threads
  - The thread that executes the kernel above sees the `aTile` declaration and understands that all its sibling-threads in the block are going to see it too. They share this variable collectively.

- The same thread, when it sees the variable `row` it understands that it has sole ownership of this variable (variable stored in a register)
3 Ways to Set Aside Shared Memory

● First way: Statically, declared inside a kernel
  ● See previous slide…

● Second way: Through the execution configuration
  ● Not that common
  ● \textbf{Ns} below indicates size (in bytes) to be allocated in shared memory

\begin{verbatim}
__global__ void MyFunc(float*) // __device__ or __global__ function
{
    extern __shared__ float shMemArray[];
    // Size of shMemArray determined through the execution configuration
    // You can use shMemArray as you wish here…
}
\end{verbatim}

// invoke like this
MyFunc<<< Dg, Db, Ns >>>(parameter);

● Third way: Dynamically, through the CUDA Driver API
  ● Advanced feature, uses API function \texttt{cuFuncSetSharedSize()}, not discussed here
Shared Memory Architecture

- Common sense observation: in a parallel machine many threads access memory at the same time
  - To service more than one thread, memory is divided into independent banks
  - This layout essential to achieve high bandwidth

- Each SM has ShMem organized in 32 Memory banks

- Recall that shared memory and L1 cache draw on the same physical memory inside an SM; i.e., they combine for 64 KB
  - This physical memory can be partitioned as
    - 48 KB of ShMem and 16 KB of L1 cache
    - The other way around
  - Note: shared memory can store less data than the registers (48 KB vs. 128 KB)
Shared Memory Architecture

- The 32 banks of the Shared Memory are organized like benches in a movie theater
  - You have multiple rows of benches
  - Each row has 32 benches
  - In each bench you can “seat” a family of four bytes (32 bits total)
  - Note that a bank represents a column of benches in the movie theater, which is perpendicular to the screen

- Each bank has a bandwidth of 32 bits per two clock cycles
Shared Memory: Transaction Rules & Bank Conflicts

- When reading in four-byte words, 32 threads in a warp attempt to access shared memory simultaneously.

- Bank conflict: the scenario where two different threads access *different* words in the same bank.

- Note that there is no conflict if different threads access any bytes within the same word.

- Bank conflicts enforce the hardware to serialize your ShMem access, which adversely impacts bandwidth.
Shared Memory Bank Conflicts

- If there are no bank conflicts:
  - Shared memory access is fast, but not as fast as register access
  - On the bright side, latency is roughly 100x lower than global memory latency

- Share memory access, the fast case:
  - If all threads of a warp access different banks, there is no bank conflict
  - If all threads of a warp access an identical address for a fetch operation, there is no bank conflict (broadcast)

- Share memory access, the slow case:
  - Worst case: 32 threads access 32 different words in the same bank
  - Must serialize all the accesses
  - In general, cost = max # of simultaneous accesses to a single bank
How Addresses Map to Banks on Fermi

- Successive 32-bit word addresses are assigned to successive banks

- Bank you work with = (address of offset) % 32
  - This is because Fermi has 32 banks
  - Example: 1D shared mem array, myShMem, of 1024 floats
    - myShMem[4]: accesses bank id #4 (relative row offset: 0)
    - myShMem[31]: accesses bank id #31 (relative row offset: 0)
    - myShMem[50]: access bank id #18 (relative row offset: 1)
    - myShMem[128]: access bank id #0 (relative row offset: 4)
    - myShMem[178]: access bank id #18 (relative row offset: 5)
  - NOTE: If, for instance, the third thread in a warp accesses myShMem[50] and the eighth thread in the warp access myShMem[178], then you have a two-way bank conflict and the two transactions get serialized

- IMPORTANT: There is no such thing as “bank conflicts” between threads belonging to different warps
Bank Addressing Examples
Transactions Involving 4 Byte Words

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

Transactions Involving 4 Byte Words

- 2-way Bank Conflicts
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 31

- 8-way Bank Conflicts
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 31
Other Examples

- Two “no conflict read” scenarios:
  - Broadcast: all threads in a warp access the same word in a bank
  - Multicast: several threads in a warp access the same word in the same bank
Data types and bank conflicts

- No conflicts below if `shrd` is a 32-bit data type:
  
  ```
  foo = shrd[baseIndex + threadIdx.x]
  ```

- Also if accessing one byte/thread, no conflict since *different* bytes of the same word are accessed
  
  - No conflicts:
    ```
    extern __shared__ char shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```

  - No conflicts:
    ```
    extern __shared__ short shrd[];
    foo = shrd[baseIndex + threadIdx.x];
    ```
Exercise: Is ShMem access below good or bad?

- Each thread loads two **floats** into shared memory:
  
  ```c
  int tid = threadIdx.x;
  sharedVar[2*tid   ] = globalVar[2*tid   ];
  sharedVar[2*tid+1] = globalVar[2*tid+1];
  ```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic
  - Doesn’t make sense in shared memory usage where there is no cache line effects but banking effects
  - 2-way-interleaved loads result in 2-way bank conflicts

- Adding insult to injury: you don’t have coalesced global memory loads – basically you are halving the device memory bandwidth
Linear Addressing

- Given:
  ```c
  __shared__ float sharedM[256];
  float foo = sharedM[baseIndex + s * threadIdx.x];
  ```

- This is bank-conflict-free if `s` shares no common factors with the number of banks
  - Conclusion: you are fine if `s` is odd
The Math Beyond Bank Conflicts

- We are in a half-warp, and the question is if thread \( t_1 \) and thread \( t_2 > t_1 \) might access the same bank of shared memory.
- Let \( b \) be the base of the array (the “shareM” pointer on previous slide).
- How should you not choose \( s \)?

\[
\begin{cases}
    b + s t_2 = b + s t_1 + 32k, \text{ for some positive integer } k \\
    0 < t_2 - t_1 \leq 32
\end{cases}
\]

\[
\begin{cases}
    32k = s(t_2 - t_1) \\
    0 < t_2 - t_1 \leq 32
\end{cases}
\]

- If \( s=2 \), take \( k=1 \), and then any threads \( t_1 \) and \( t_2 \) which are 16 apart satisfy the condition above and will have a bank conflict ([0,16], [1,17], etc.) – two way conflict.
- If \( s=4 \), take \( k=2 \), any threads \( t_1 \) and \( t_2 \) which are 8 apart will have a bank conflict ([0, 8,16,24], [1,9,17,25], etc.) – four way conflict.
- NOTE: you can’t get a bank conflict is \( s \) is odd (no quartet \( k, s, t_1, t_2 \) satisfies the bank conflict condition above). So take stride \( s=1,3,5 \), etc.
Example, ShMem Use: Vector Reduction

- Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads participate
- The process is memory bound, low arithmetic ratio…
- Covered in more detail on Th (also part of the Assignment)
  - Used as a vehicle to demonstrate CUDA optimization techniques

Data staged in shared memory
A small number of threads finishes off
Example: Vector Reduction with Bank Conflicts
(assume 2048 vector entries stored in shared memory; one block (1024 threads) carries out the reduction)
Vector Reduction **without** Bank Conflicts
(assume 2048 vector entries stored in shared memory; one block (1024 threads) carries out the reduction)
Shared Memory: A Word of Caution

- It used to be that any access to Shared Memory was a direct access (in compute capability 1.x)

- Fermi (2.x) has a load/store architecture that can bring data into registers
  - This means that there is no guarantee for coherence between the shared memory block and the value stored in the register

- Problem is typically addressed by making that shared memory volatile:
  - In 1.x, this was always ok:
    ```c
    __shared__ int myShVars[256];
    ```
  - In 2.x, you might have to do this (the compiler doesn’t optimize instructions related to `myShVars`):
    ```c
    volatile __shared__ int myShVars[256];
    ```

More information about shared memory: Programming Guide, Sections 3.2.3, 5.3.2.3, and Appendix F4.3
Example: Is 48KB of Shared Memory Enough?

[Revisiting the Matrix Multiplication Example]

- One block computes one tile $C_{sub}$ of size $Block_{Size}$

- One thread computes one element of $C_{sub}$

- Assume that the dimensions of $A$ and $B$ are multiples of $Block_{Size}$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
Example: Matrix Multiplication
Shared Memory Usage - WIDTH = 16

- Each Block requires $2 \times WIDTH^2 \times 4$ bytes of shared memory storage
  
  - For WIDTH = 16, each BLOCK requires 2KB, up to 24 Blocks can fit into the Shared Memory of GTX480
    - Note that if you have the setting for ShMemory that gives you 16 KB you can still fit 8 blocks on one SM

  - Since each SM scheduler on GTX480 can only manage 1536 threads (48 warps), each SM can only take 6 Blocks of 256 threads each
    - Then, you have 100% occupancy

- Shared memory size is not a constraint for our implementation of the Matrix Multiplication
Example: Matrix Multiplication
Shared Memory Usage - WIDTH = 32

- Each Block requires $2 \times WIDTH^2 \times 4$ bytes of shared memory storage

  - For WIDTH = 32, each BLOCK requires 8KB, up to 6 Blocks can fit into the Shared Memory of GTX480
    - Note that if you have the setting for ShMemory that gives you 16 KB you can still fit 2 blocks on one SM

  - Since each SM on GTX480 can only manage 1536 threads, each SM can only take 1 Block of 1024 threads
    - Then, you have 66% occupancy

- Conclusion: It’s likely that this will run slower than the WIDTH=16 options
  - Not necessarily true, since there are other factors (number of registers used, potential for compiler code optimization, etc.) that come into play
Synchronization Issues
Global Communication

- Keep this in mind: in this segment we are not talking about execution scheduling
  - Focus is on making data available to other threads and required synchronization

- How can Global Communication occur?
  - For threads in different blocks and different grids:
    - Locations in global memory (global variables)
  
  - For threads in same blocks:
    - Locations in global memory
    - Locations in shared memory (\texttt{__shared__} variables)

- Looming danger: race conditions…
Race Conditions

Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is a write.

```c
// race.cu, the kernel
__global__ void race(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    *x = i;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudamemcpy(x, d_x, sizeof(int), cudamemcpyDeviceToHost);
```
Race Conditions

Programs with race conditions may produce unexpected, seemingly arbitrary results
- Updates may be missed, and updates may be lost

```c
// race.cu, the kernel
__global__ void race(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    *x = *x + 1;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization for Data Communication

- Accesses to shared locations need to be correctly synchronized (coordinated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data.

- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.

- Whenever possible, try hard to design algorithms with few synchronizations.
  - Synchronization impacts execution speed.
ME759
High Performance Computing for Engineering Applications

Atomic Operations
Profiling CUDA Code

October 9, 2013

"In God we trust, all others bring data."
- W. Edwards Deming
Before We Get Started…

- Last time
  - Shared memory – bank conflicts issues
  - Started “Synchronization for Data Communication under CUDA”

- Today
  - Atomic operations (part of “Synchronization for Data Communication”)
  - Profiling CUDA code

- Miscellaneous
  - Fifth assignment posted online. GPU computing related and challenging
  - Please provide feedback
    - I’ll compile all of your feedback and upload on the class website for general access

- Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8)
  - Review session on Wd, Nov. 6 @ 6 PM in this room (2121ME)
  - Exam will draw on material covered in class and information provided in the primer
  - It’ll be a pen and paper exam. Open book and open anything
Choreographing Memory Operations

- Accesses to shared locations (global memory & shared memory) need to be correctly synchronized (coordinated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data.

- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.

- Whenever possible, try hard to design algorithms with few synchronizations.
  - Coordination between threads impacts execution speed.
Don’t Do This at Home

- Assume thread T1 reads a value defined by thread T0

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Program needs to ensure that thread T1 reads location after thread T0 has written location
Synchronization within Block

- Threads in same block: can use `__syncthreads()` to specify synchronization point that orders accesses

```c
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x;
    if (i == 0) *x = blockIdx.x;
    __syncthreads();
    if (i == 1) *y = *x;
}
```

```c
// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Here’s a fun question: would this work if the kernel is launched with an execution configuration that has two blocks?
Synchronization between Grids

- Threads in different grids: system ensures writes from kernel happen before reads from subsequent grid launches.

```c
// update.cu
__global__ void update_x(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0) *x = 1;
}

__global__ void update_y(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update_x<<<1,2>>>(d_x, d_y);
update_y<<<1,2>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Synchronization within Grid
[The Need for Atomics]

- Often not reasonable to split kernels to synchronize reads and writes from different threads to common locations. Here’re two reasons:
  - Values of \_\_\_\_shared\_\_\_ variables are lost unless explicitly saved
  - Kernel launch overhead is nontrivial – extra launches can degrade performance

- CUDA provides atomic functions (commonly called atomic memory operations) to enforce atomic accesses to shared variables that may be accessed by multiple threads

- Programmers can synthesize various coordination objects and synchronization schemes using atomic functions.
Atomics
Atomics, Introduction

- Atomic memory operations (atomic functions) are used to solve coordination problems in parallel computer systems.

- General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.
Atomic Functions

Atomic functions perform read-modify-write operations on data residing in global and shared memory.

```c
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, 1);    // j = *x;
}
```

// snippet of code in main.cpp
```c
int x = 0;
cudaMemcpy(&d_x, &x, cudaMemcpyHostToDevice);
update<<<1,128>>>(x_d);
cudaMemcpy(&x, &d_x, cudaMemcpyDeviceToHost);
```

Atomic functions guarantee that only one thread may access a memory location while the operation completes.

Order in which threads get to write is not specified though…
Atomic Functions

Atomic functions perform read-modify-write operations on data that can reside in global or shared memory.

Synopsis of atomic function $\text{atomicOP}(a, b)$ is typically

```c
    t1 = *a;    // read
    t2 = t1 OP (*b); // modify
    *a = t2;    // write
    return t1;
```

- The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.
- The atomic function returns the initial value, not the final value, stored at the memory location.
Atomic Functions

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.

- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.

- However, none of the atomic updates will be lost.

- Many different kinds of atomic operations:
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or), Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap
A Histogram Example

// Compute histogram of colors in an image
//
//  color – pointer to picture color data
//  bucket – pointer to histogram buckets, one per color
//
__global__ void histogram(int n, int* color, int* bucket)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
        int c = colors[i];
        atomicAdd(&bucket[c], 1);
    }
}
/ For algorithms where the amount of work per item
/ is highly non-uniform, it often makes sense
/ to continuously grab work from a queue

__device__ int do_work(int x)
{
    return f(x-1) + f(x) + f(x+1);
}

__global__ void process_work_q(int* work_q, int* q_counter,
    int* output, int queue_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, 1);
    if(q_index<queue_max) {
        int result = do_work(work_q[q_index]);
        output[i] = result;
    }
}
Performance Notes

- Atomics are slower than normal accesses (loads, stores)

- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations

- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location

- Atomics: convenient to use, come at a typically high efficiency loss…
Example: Global Min/Max (Naive)

- Compute maximum across all threads in a grid
- One can use a single global maximum value, but it will be VERY slow

```c
__global__ void global_max(int* values, int* global_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(global_max, val);
}
```
Example: Global Min/Max (Better)

- Introduce local maximums and update global only when new local maximum found

```c
__global__ void global_max(int* values, int* global_max,
                          int *local_max, int num_locals)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    int li = i % num_locals;
    int old_max = atomicMax(&local_max[li], val);
    if (old_max < val)
    {
        atomicMax(global_max, val);
    }
}
```

- Reduces frequency at which threads attempt to update the global maximum, reducing competition access to location

NVIDIA [J. Balfour]→
Lessons from global Min/Max

- Many updates to a single value causes serial bottleneck

- One can create a hierarchy of values to introduce more parallelism and locality into algorithm

- However, performance can still be slow, so use judiciously
Important note about Atomics

- Atomic updates are not guaranteed to appear atomic to concurrent accesses using loads and stores

```c
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x = j + i;
    }
}

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
```
Summary of Atomics

- When to use: Cannot use normal load/store for reliable inter-thread communication because of race conditions
- Use atomic functions for infrequent, sparse, and/or unpredictable global communication
- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism
- Attempt to use shared memory and structure algorithms to avoid synchronization whenever possible
CUDA: Measuring Speed of Execution
[Gauging Greatness]
“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.”

Donald Knuth

In “Structured Programming With Go To Statements” Computing Surveys, Vol. 6, No. 4, December 1974
Available on class website.
Next, the discussion focuses on tools you can use to find that 3% of the code worth optimizing…
Code Timing/Profiling

-Lazy man’s solution
  - Do nothing, instruct the executable to register crude profiling info

-Advanced approach: use NVIDIA’s `nvvp` Visual Profiler
  - Visualize CPU and GPU activity
  - Identify optimization opportunities
  - Allows for automated analysis
  - `nvvp` is a cross platform tool (linux, mac, windows)
Lazy Man’s Solution…

- Set the right environment variable and run your executable [illustrated on Euler]:

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> export CUDA_PROFILE=1
>> ./testV4_20
>> cat cuda_profile_0.log
```

```plaintext
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```
Lazy Man’s Solution…

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR ffffffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpvHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[__Z14applyStencillDiiPKfPfS1__] gputime=[ 166.944 ] cputime=[ 13.000 ] occupancy=[1.0]
method=[ memcpvDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]

>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpvHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[__Z14applyStencillDiiPKfPfS1__] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpvDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Lazy Man’s Solution...

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]

>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ __Z14applyStencillDiiPKfPfS1__ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```
Lazy Man’s Solution...

```bash
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR fffff6c689a404a8
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]
```

Compute capability 2.0 (Fermi)

```bash
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method, gputime, cputime, occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
```

Compute capability 1.0 (Tesla/G80)
**nvvp: NVIDIA Visual Profiler**

- Available on Euler
- Provides a nice GUI and ample information regarding your run
- Many bells & whistles
  - Covering here the basics through a 1D stencil example
- Acknowledgement: Discussion on *nvvp* uses material from NVIDIA (S. Satoor).
  - Slides that include this material marked by “NVIDIA [S. Satoor]→” sign at bottom of slide
1D Stencil: A Common Algorithmic Pattern
[Problem Used to Introduce Profiling Tool]

- Applying a 1D stencil to a 1D array of elements
  - Function of input elements within a radius

- Fundamental to many algorithms
  - Standard discretization methods, interpolation, convolution, filtering,…

- This example will use weighted arithmetic mean
Serial Algorithm

\(\because\) = CPU Thread

(radius = 3)

\(f\)

NVIDIA [S. Satoor]→
Serial Algorithm

\( \therefore = \text{CPU Thread} \)

\( \text{(radius = 3)} \)

Repeat for each element
Serial Implementation

```c
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS, N-RADIUS, weights, in, out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS, N - RADIUS, weights, in, out);
    // free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);
    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);

    applyStencil1D(RADIUS,N-RADIUS,weights,in,out);

    //free resources
    free(weights); free(in); free(out);
}

void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    for (int i = sIdx; i < eIdx; i++) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}

<table>
<thead>
<tr>
<th>CPU</th>
<th>MEElements/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930</td>
<td>30</td>
</tr>
</tbody>
</table>

NVIDIA [S. Satoor] →
Parallel Algorithm

**Serial**: One element at a time

in

... ... ...

... out↳ ...

**Parallel**: Many elements at a time

in

... ... ...

... out↳ ...

↳ = Thread

NVIDIA [S. Satoor]→
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if ( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float*out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

_global_ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

Copy

GPU

Inputs
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    // free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
The Parallel Implementation

```c
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);

    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    //free resources
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}
```

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

Copy results from GPU

NVIDIA [S. Satoor]→
void main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    // allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights;  cudaMalloc(&d_weights, wsize);
    float *d_in;       cudaMalloc(&d_in, size);
    float *d_out;      cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights,weights,wsize,cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    free(weights); free(in); free(out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
}

Device | Algorithm          | MEElements/s | Speedup |
--------|--------------------|--------------|---------|
i7-930* | Optimized & Parallel | 130          | 1x      |
Tesla C2075 | Simple             | 285          | 2.2x    |

__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if( i < eIdx ) {
        out[i] = 0;
        // loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
Application Optimization Process

[Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - `cuda-gdb`, `cuda-memcheck`

- Optimize
  - …dealing with this next
NVIDIA Visual Profiler

Timeline of CPU and GPU activity

Kernel and memcpy details

NVIDIA [S. Satoor]
NVIDIA Visual Profiler

CUDA API activity on CPU

Memcpy and kernel activity on GPU
Detecting Low Memory Throughput

- Spend majority of time in data transfer
  - Often can be overlapped with preceding or following computation

- From timeline can see that throughput is low
  - PCIe x16 can sustain > 5GB/s
How do we know when there is an optimization opportunity?
- Timeline visualization seems to indicate an opportunity
- Documentation gives guidance and strategies for tuning
  - CUDA Best Practices Guide – link on the website
  - CUDA Programming Guide – link on the website

Visual Profiler analyzes your application
- Uses timeline and other collected information
- Highlights specific guidance from Best Practices
- Like having a customized Best Practices Guide for your application
Visual Profiler Analysis

Several types of analysis are provided

Analysis pointing out low memcpy throughput
Online Optimization Help

LowMemcpy Throughput [ 997.19 MB/s avg, for memcyps accounting for 68.1% of all memcpy time ]
The memory copies are not fully using the available host to device bandwidth.

Each analysis has link to Best Practices documentation

Pinned Memory

Page-locked or pinned memory transfers attain the highest bandwidth between the host and the device. On PCIe x16 Gen2 cards, for example, pinned memory can attain greater than 5 GBps transfer rates.

Pinned memory is allocated using the cudaMemcpyHost() or cudaMemcpy() functions in the Runtime API. The bandwidthTest.cu program in the CUDA SDK shows how to use these functions as well as how to measure memory transfer performance.

Pinned memory should not be overused. Excessive use can reduce overall system performance because pinned memory is a scarce resource. How much is too much is difficult to tell in advance, so as with all optimizations, test the applications and the systems they run on for optimal performance parameters.

Parent topic: Data Transfer Between Host and Device

NVIDIA [S. Satoor]
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights; cudaMallocHost(&weights, wsize);
    float *in; cudaMallocHost(&in, size);
    float *out; cudaMallocHost(&out, size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights);
    float *d_in; cudaMalloc(&d_in);
    float *d_out; cudaMalloc(&d_out);
    ...

    //CPU allocations use pinned memory to enable fast memcpy
    //No other changes
}
Pinned CPU Memory Result

NVIDIA [S. Satoor]→
# Pinned CPU Memory Result

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
</tbody>
</table>

*4 cores + hyperthreading
Application Optimization Process

Revisited

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

- Optimize
  - Profiler (pinned memory)
Application Optimization Process [Revisited]

- Identify Optimization Opportunities
  - 1D stencil algorithm

- Parallelize with CUDA, confirm functional correctness
  - Debugger
  - Memory Checker

Optimize
  - Profiler (pinned memory)
- Advanced optimization
  - Larger time investment
  - Potential for larger speedup

Asynchronous Transfers and Overlapping Transfers with Computation

Data transfers between the host and the device using cudaMemcpy() are blocking transfers; that is, control is returned to the host thread only after the data transfer is complete. The cudaMemcpyAsync() function is a non-blocking variant of cudaMemcpy() in which control is returned immediately to the host thread. In contrast with cudaMemcpy(), the asynchronous transfer version requires pinned host memory (see Pinned Memory), and it contains an additional argument, a stream ID. A stream is simply a sequence of operations that are performed in order on the device. Operations in different streams can be interleaved and in some cases overlapped—a property that can be used to hide data transfers between the host and the device.

Asynchronous transfers enable overlap of data transfers with computation in two different ways. On all CUDA-enabled devices, it is possible to overlap host computation with asynchronous data transfers and with device computations. For example, Overlapping computation and data transfers demonstrates how host computation in the
Data Partitioning Example

 Partition data into TWO chunks

 chunk 1

 in

 chunk 2

 out

 NVIDIA [S. Satoor]
Data Partitioning Example

chunk 1

memcpy
compute
memcpy

chunk 2

in

memcpy
compute
memcpy

out
Data Partitioning Example

chunk 1

chunk 2

memcpy compute memcpy compute memcpy

NVIDIA [S. Satoor]
Overlapped Compute/Memcpy

[problem broken into 16 chunks]
Overlapped Compute/Memcpy

Compute time completely “hidden”

Exploit dual memcpy engines

NVIDIA [S. Satoor]
## Overlapped Compute/Memcpy

<table>
<thead>
<tr>
<th>Device</th>
<th>Algorithm</th>
<th>MEElements/s</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-930*</td>
<td>Optimized &amp; Parallel</td>
<td>130</td>
<td>1x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Simple</td>
<td>285</td>
<td>2.2x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Pinned Memory</td>
<td>560</td>
<td>4.3x</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>Overlap</td>
<td>935</td>
<td>7.2x</td>
</tr>
</tbody>
</table>

**Diagram:**
- Process: 8689
- Thread: 812144512
- Runtime API
- Driver API
- [0] Tesla C2075
- Context 1 (CUDA)

**Graph:**
- cudaDeviceSynchronize
- ME759: Use of multiple streams covered in a week
High Performance Computing for Engineering Applications

Optimizing CUDA Code

October 11, 2013

“Attitude is a little thing that makes a big difference.”
-- Winston Churchill
Before We Get Started…

- Last time
  - Atomic operations (part of “Synchronization for Data Communication”)
  - Profiling CUDA code

- Today: more of an hands-on lecture
  - Tiling as a programming pattern to speed up CUDA code
  - Simple example of finding bugs and improving performance: stencil operation
  - More complex example of optimizing code: vector reduction on the GPU (like your HW)

- Miscellaneous
  - Feedback has been uploaded on the course website
  - A webpage is available that reports the use of the HW extension (see my email of last night)
  - Email to describe rules of engagement for Midterm project coming your way soon
  - Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8). Room: 1153ME
    - Review session on Wd, Nov. 6 @ 6 PM in this room (2121ME)
    - Exam will draw on material covered in class and information provided in the primer
    - It’ll be a pen and paper exam. Open book and open anything

Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8). Room: 1153ME
Optimization Summary

[Looking Back at 1D Stencil Example…]

- Initial CUDA parallelization
  - Expeditious, kernel is almost word-for-word replica of sequential code
  - 2.2x speedup

- Optimize memory throughput
  - Expeditious, need to know about pinned memory
  - 4.3x speedup

- Overlap compute and data movement
  - More involved, need to know about the inner works of CUDA
  - Problem should be large enough to justify mem-transfer/execution
  - 7.2x speedup
Iterative Optimization

- Identify Optimization Opportunities
- Parallelize
- Optimize
Take Home Message…

- Regard CUDA as a way to accelerate the compute-intensive parts of your application

- Visual profiler (nvpp) helps in performance analysis and optimization
Revisit Stencil Example

- Problem setup
  - 1,000,000 elements
  - RADIUS is 3

- Purpose:
  - Show a typical bug and then one easy way to get some extra performance out of the code
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    float *cuda_out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<N/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);

    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: "<< nDiffs << " differences\n";

    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
Example: Debugging & Profiling
[1DStencil Code: Supporting Cast]

```c
int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=0; i<nElements; i++)
        if( fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}
```

```c
void initializeWeights(float* weights, int rad) {
    // for now hardcoded for RADIUS=3
    weights[0] = 0.50f;
    weights[1] = 0.75f;
    weights[2] = 1.25f;
    weights[3] = 2.00f;
    weights[4] = 1.25f;
    weights[5] = 0.75f;
    weights[6] = 0.50f;
}
```

```c
void initializeArray(float* arr, int nElements) {
    const int myMinNumber = -5;
    const int myMaxNumber = 5;
    srand(time(NULL));
    for(int i=0; i<nElements; i++)
        arr[i] = (float)(rand() % (myMaxNumber - myMinNumber + 1) + myMinNumber);
}
```
Example: Debugging & Profiling
[1DStencil Code: the actual stencil function]
First Version...

[negrut@euler CodeBits]$ qsub -I -l nodes=1:gpus=1:default -X
[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV1.cu
[negrut@euler01 CodeBits]$ ./testV1
Doesn't look good: 57 differences
[negrut@euler01 CodeBits]$
Example: Debugging & Profiling

[1DStencil Code]

```c
int main() {
    int size = N * sizeof(float);
    int wsize = (2 * RADIUS + 1) * sizeof(float);
    //allocate resources
    float *weights = (float *)malloc(wsize);
    float *in = (float *)malloc(size);
    float *out = (float *)malloc(size);
    float *cuda_out= (float *)malloc(size);
    initializeWeights(weights, RADIUS);
    initializeArray(in, N);
    float *d_weights; cudaMalloc(&d_weights, wsize);
    float *d_in; cudaMalloc(&d_in, size);
    float *d_out; cudaMalloc(&d_out, size);
    cudaMemcpy(d_weights, weights, wsize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    applyStencil1D<<<(N+511)/512, 512>>>(RADIUS, N-RADIUS, d_weights, d_in, d_out);
    applyStencil1D_SEQ(RADIUS, N-RADIUS, weights, in, out);
    cudaMemcpy(cuda_out, d_out, size, cudaMemcpyDeviceToHost);
    int nDiffs = checkResults(cuda_out, out, N);
    nDiffs==0? std::cout<<"Looks good.\n": std::cout<<"Doesn't look good: "<< nDiffs << " differences\n";
    //free resources
    free(weights); free(in); free(out); free(cuda_out);
    cudaFree(d_weights); cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```
Second Version...

[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV2.cu
[negrut@euler01 CodeBits]$ ./testV2

Doesn't look good: 4 differences

[negrut@euler01 CodeBits]$

- Reason: checkResults runs a loop over all 1,000,000 entries. It should exclude the first RADIUS and last RADIUS of them... Those entries are not computed, you pick up whatever was there when memory was allocated on the host and on the device. As such, it gives false positives

- NOTE: this problem is not reproducible always (sometimes code runs ok, sometimes gives you a false positive)
int checkResults(float* cudaRes, float* res, int nElements) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=0; i<nElements; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}

int checkResults(int startElem, int endElem, float* cudaRes, float* res) {
    int nDiffs=0;
    const float smallVal = 0.000001f;
    for(int i=startElem; i<endElem; i++)
        if(fabs(cudaRes[i]-res[i])>smallVal )
            nDiffs++;
    return nDiffs;
}
Third Version [V3]...

[negrut@euler01 CodeBits]$ nvcc -gencode arch=compute_20, code=sm_20 testV3.cu
[negrut@euler01 CodeBits]$ ./testV3
Looks good.

[negrut@euler01 CodeBits]$

- Things are good now...
Code Profiling...

- Code looks like running ok, no evident bugs

- Time to profile the code, we’ll use the Lazy Man’s approach

- Profile V3 version
  - Create base results, both for compute capability 1.0 (Tesla) and 2.0 (Fermi)
Lazy Man’s Solution...

Compute capability 2.0 (Fermi)

```
>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV3.cu -o testV3_20
>> ./testV3_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR fffff6c689a59e98
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1.664 ] cputime=[ 9.000 ]
method=[ memcpyHtoD ] gputime=[ 995.584 ] cputime=[ 1193.000 ]
method=[ __Z14applyStencil1DiPKfPfS1_ ] gputime=[ 189.856 ] cputime=[ 12.000 ] occupancy=[1.0]
method=[ memcpyDtoH ] gputime=[ 1977.728 ] cputime=[ 2525.000 ]
```

Compute capability 1.0 (Tesla/G80)

```
>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV3.cu -o testV3_10
>> ./testV3_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b1842064
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1787.232 ] cputime=[ 2760.139 ]
method=[ __Z14applyStencil1DiPKfPfS1_ ] gputime=[ 68357.69 ] cputime=[ 8.85 ] occupancy=[0.667]
```
Improving Performance

- Here’s what we’ll be focusing on:

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) { out[i] = 0;
        //loop over all elements in the stencil
        for (int j = -RADIUS; j <= RADIUS; j++) {
            out[i] += weights[j + RADIUS] * in[i + j];
        }
        out[i] = out[i] / (2 * RADIUS + 1);
    }
}
```

- There are several opportunities for improvement to move from V3 to V4:
  - Too many accesses to global memory (an issue if you don’t have L1 cache)
  - You can unroll the 7-iteration loop (it’ll save you some pocket change)
  - You can use shared memory (important if you don’t have L1 cache, i.e., in 1.0)
  - You can use pinned host memory [you have to look into `main()` to this end]
Improving Performance [V4]

- Version V4: Take care of
  - Repeated access to global memory
  - Loop unrolling

```c
__global__ void applyStencil1D(int sIdx, int eIdx, const float *weights, float *in, float *out) {
    int i = sIdx + blockIdx.x*blockDim.x + threadIdx.x;
    if (i < eIdx) {
      float result = 0.f;
      result += weights[0]*in[i-3];
      result += weights[1]*in[i-2];
      result += weights[2]*in[i-1];
      result += weights[3]*in[i];
      result += weights[4]*in[i+1];
      result += weights[5]*in[i+2];
      result += weights[6]*in[i+3];
      result /= 7.f;
      out[i] = result;
    }
}
```

- Even now there is room for improvement
  - You can have `weights` and `in` stored in shared memory
  - You can use pinned memory (mapped memory) on the host
Lazy Man’s Profiling: V4

>> nvcc -O3 -gencode arch=compute_20,code=sm_20 testV4.cu -o testV4_20
>> ./testV4_20

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GTX 480
# TIMESTAMPFACTOR ffffffff6c689a404a8
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1001.952 ] cputime=[ 1197.000 ]
method=[ memcpyDtoH ] gputime=[ 1394.144 ] cputime=[ 2533.000 ]

>> nvcc -O3 -gencode arch=compute_10,code=sm_10 testV4.cu -o testV4_10
>> ./testV4_10

# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 GeForce GT 130M
# TIMESTAMPFACTOR 12764ee9b183e71e
method,gputime,cputime,occupancy
method=[ memcpyHtoD ] gputime=[ 1815.424 ] cputime=[ 2787.856 ]
method=[ _Z14applyStencil1DiiPKfPfS1_ ] gputime=[ 47332.9 ] cputime=[ 8.469 ] occupancy=[0.67]
method=[ memcpyDtoH ] gputime=[ 3535.648 ] cputime=[ 4555.577 ]
# Timing Results

[Two Different Approaches (V3, V4) & Two Different GPUs (sm_20, sm_10)]
[each executable was run 7 times; script available on the class website]

<table>
<thead>
<tr>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>166.752</td>
<td>190.560</td>
<td>47341.566</td>
<td>68611.008</td>
</tr>
<tr>
<td>166.912</td>
<td>190.016</td>
<td>47332.930</td>
<td>68531.875</td>
</tr>
<tr>
<td>166.976</td>
<td>190.208</td>
<td>47391.039</td>
<td>68674.109</td>
</tr>
<tr>
<td>166.368</td>
<td>190.048</td>
<td>47252.734</td>
<td>68679.422</td>
</tr>
<tr>
<td>166.848</td>
<td>189.696</td>
<td>47371.426</td>
<td>68357.695</td>
</tr>
<tr>
<td>166.592</td>
<td>189.856</td>
<td>47250.465</td>
<td>68618.492</td>
</tr>
<tr>
<td>166.944</td>
<td>190.240</td>
<td>47379.902</td>
<td>68687.266</td>
</tr>
</tbody>
</table>

**Averages**

<table>
<thead>
<tr>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>166.7702857</td>
<td>190.0891429</td>
<td>47331.43743</td>
<td>68594.26671</td>
</tr>
</tbody>
</table>

**Standard Deviations**

<table>
<thead>
<tr>
<th>V4_20</th>
<th>V3_20</th>
<th>V4_10</th>
<th>V3_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.132410266</td>
<td>0.147947777</td>
<td>0.123060609</td>
<td>0.171466201</td>
</tr>
</tbody>
</table>

**Slowdown, sm_20**  
13.98262109%

**Slowdown, sm_10**  
44.92326969%
This is how you should thing about code profiling and optimization:

- Would you ever send out your CV right after you completed writing it?
- Probably not, you always go back and spend a bit of time polishing it…
Putting Things in Perspective…

Here’s what we’ve covered so far:
- CUDA execution configuration (grids, blocks, threads)
- CUDA scheduling issues (warps, thread divergence, synchronization, etc.)
- CUDA Memory ecosystem (registers, shared mem, device mem, L1/L2 cache, etc.)
- Practical things: building, debugging, profiling CUDA code

Next: CUDA GPU Programming - Examples & Code Optimization Issues
- Tiling: a CUDA programming pattern
- Example: CUDA optimization exercise in relation to a vector reduction operation
- CUDA Execution Configuration Optimization Heuristics: Occupancy issues
- CUDA Optimization Rules of Thumb
Tiling [Blocking]: A Fundamental CUDA Programming Pattern

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a block of threads
  - No different from cache blocking!
    - Except you now have full control over it

- Provides several significant performance benefits
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory
Fundamental CUDA Pattern: Tiling

- **Partition** data into subsets that fit into **__shared__** memory

This is your data: one big chunk, about to be broken into subsets suitable to be stored into shared memory.
Fundamental CUDA Pattern: Tiling

- Process each data subset with one **thread block**
Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
Fundamental CUDA Pattern: Tiling

- Perform the computation on the subset from shared memory
Fundamental CUDA Pattern: Tiling

- Copy the result from **shared** memory back to global memory
A large number of CUDA kernels are built this way.

However, tiling [blocking] may not be the only approach to solving a problem, sometimes it might not apply…

Two questions that can guide you in deciding if tiling is it:

- Does a thread require several loads from global memory to serve its purpose?
- Could data used by a thread be used by some other thread in the same block?
- If answer to both questions is “yes”, consider tiling as a design pattern.

The answer to these two questions above is not always obvious.

- Sometime it’s useful to craft an altogether new approach (algorithm) that is capable of using tiling: you force the answers to be “yes”.
A CUDA Optimization Exercise
[A Demonstration Using the Parallel Reduction Application]
Parallel Reduction in CUDA

- Exercise draws on material made available by Mark Harris of NVIDIA
  [acknowledgement at bottom of slides]

- Parallel Reduction: Common and very important data parallel primitive
  - Example: Used to compute the norm of a large vector

- Easy to implement in CUDA
  - Challenging to get it to run fast though

- Serves as a good optimization example
  - Walk step by step through several different versions
  - Demonstrates several important optimization strategies
Parallel Reduction

- Basic Idea: tree-based approach used within each thread block

- Need to be able to use multiple thread blocks
  - Why? To process very large arrays
  - Why? To keep all multiprocessors on the GPU busy
  - How? Each thread block reduces a portion of the array to one single value

- Q: How do we communicate partial results between thread blocks?
Problem: Global Synchronization

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively

- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than number of SMs times the number of resident blocks / SM) → this may reduce overall efficiency

- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead
Multiple Kernel Calls

[An Example, and how it all works out…]

- Imagine you launch a grid in which each block has 256 threads.

- Assume that the number or elements in the array is \( N = 100,000 \)
  - Note that \( 100,000 = 390 \times 256 + 160 \), therefore \( \text{ceil}[N/256.0] = 391 \)

- For the first stage, you launch 391 blocks of 256 threads
  - At the end of this stage you still have to operate on 391 elements.

- For the second stage, you launch two blocks of 256 threads
  - At the end of this stage you only have to operate on two elements.

- For the third and last stage, you launch one block of 32 threads
  - Almost all threads idle…

- NOTE: after the first stage, each subsequent stage operates on a number of entries equal to the number of blocks in the previous stage.
Vector Reduction: 30,000 Feet Perspective

- At the block level: Bring data in shared memory, then start adding in parallel
- Fewer and fewer threads of a block participate
- The process is memory bound, low arithmetic intensity…
What is Our Optimization Goal?

- We should strive to reach GPU peak performance
- Choose the right metric:
  - GFLOP/s: for compute-bound kernels
  - Bandwidth: for memory-bound kernels
- Reductions have very low arithmetic intensity
  - 1 flop per element loaded (bandwidth-optimal)
- Therefore we should strive for peak bandwidth
- This example uses results generated using a G80 GPU
  - Compute capability (CC) 1.0
  - 384-bit memory interface, 900 MHz DDR
  - $384 \times 900 \times 2 / 8 = 86.4 \text{ GB/s}$
  - Example carries over to other CCs, this algorithm will be memory bound
Parallel Reduction: Interleaved Addressing

Note: in stage $s$, only threads divisible to $2^s$ get to work. Stride: $2^{(s-1)}$
Reduction #1: Interleaved Addressing

```c
__global__ void reduce1(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global memory
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Problem: highly divergent warps are very inefficient, and % operator is very slow
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time ($2^{22}$ ints)</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
</tr>
</tbody>
</table>

Note: Block Size = 128 threads for all tests
Parallel Reduction: Interleaved Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>1</th>
<th>8</th>
<th>-1</th>
<th>0</th>
<th>-2</th>
<th>3</th>
<th>5</th>
<th>-2</th>
<th>-3</th>
<th>2</th>
<th>7</th>
<th>0</th>
<th>11</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
</table>

Step 1 Stride $2^0$

Values

|   | 11 | 1  | 7  | -1 | -2 | -2 | 8  | 5  | -5 | -3 | 9  | 7  | 11 | 11 | 2  | 2  |

Thread IDs

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |

Step 2 Stride $2^1$

Values

|   | 18 | 1  | 7  | -1 | 6  | -2 | 8  | 5  | 4  | -3 | 9  | 7  | 13 | 11 | 2  | 2  |

Thread IDs

|   | 0  | 1  | 2  | 3  |

Step 3 Stride $2^2$

Values

|   | 24 | 1  | 7  | -1 | 6  | -2 | 8  | 5  | 17 | -3 | 9  | 7  | 13 | 11 | 2  | 2  |

Thread IDs

|   | 0  | 1  |

Step 4 Stride $2^3$

Values

|   | 41 | 1  | 7  | -1 | 6  | -2 | 8  | 5  | 17 | -3 | 9  | 7  | 13 | 11 | 2  | 2  |

New Problem: Shared Memory Bank Conflicts
Reduction #2: Interleaved Addressing

Just replace divergent branch in inner loop...

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

...with strided index and non-divergent branch:

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Interleaved addressing with divergent branching</th>
<th>8.054 ms</th>
<th>2.083 GB/s</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 2:</td>
<td>Interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
</tbody>
</table>
Parallel Reduction: Sequential Addressing

Sequential addressing is Shared Mem conflict free
Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop...

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

...with reversed loop and threadID-based indexing:

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel 1:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>with divergent branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel 2:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved addressing</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>with bank conflicts</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel 3:</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
</tbody>
</table>
Idle Threads...

Current solution:

```c
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

Note that half of the threads are idle on first loop iteration! This is wasteful...
Reduction #4: First Add During Load

Replace single load:

```c
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

...With two loads and first add of the reduction:

```c
// perform first level of reduction upon reading from
// global memory and writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

One side effect: the number of blocks you need now is half of what it used to be...
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4: first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
</tbody>
</table>
Instruction Bottleneck

- At 17 GB/s, we’re far from bandwidth bound
  - And we know reduction has low arithmetic intensity

- Therefore a likely bottleneck is instruction overhead
  - Ancillary instructions that are not loads, stores, or arithmetic for the core computation
  - In other words: address arithmetic and loop overhead

- Strategy: unroll loops
Unrolling the Last Warp

- As reduction proceeds, the number of “active” threads decreases
  - When \( s \leq 32 \), we have only one warp left

- Instructions are SIMD synchronous within a warp
  - All threads in a warp proceed in lockstep fashion

- That means when \( s \leq 32 \):
  - We don’t need to \_
    _syncthreads()_  
  - We don’t need “if (tid < s)” because it doesn’t save any work

- Let’s unroll the last 6 iterations of the inner loop
Note: This saves useless work in all warps, not just the last one!

Without unrolling, all warps execute every iteration of the for loop and if statement
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
</tbody>
</table>
Complete Unrolling

- If we knew the number of iterations (or equivalently, of threads in a block) at compile time, we could completely unroll the reduction
  - Luckily, the block size on G80 is limited by the GPU to 512 threads
    - 1024 on newer Fermi GPUs
  - Also, we are sticking to power-of-2 block sizes

- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don’t know at compile time?

- Use of templates can solve this issue…
  - CUDA supports C++ template parameters on device and host functions
Unrolling with Templates

- Specify block size as a function template parameter
- The kernel is parameterized:

```cpp
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata)
```
Reduction #6: Completely Unrolled

```
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

- All code in RED will be evaluated at compile time. Results in a very efficient inner loop.
- For Fermi, you’d have one more if statement that covers the case when blockSize>=1024
- You can call the warpReduce function only when you got to one wrap. Reason: you don’t have to synchronize at that point.
Invoking Template Kernels

```c
switch (threads) {
    case 512:
        reduce6<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
        reduce6<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
        reduce6<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
        reduce6< 64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
        reduce6< 32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
        reduce6< 16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
        reduce6<  8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
        reduce6<  4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
        reduce6<  2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
        reduce6<  1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
}
```
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2(^{22}) ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
</tbody>
</table>
Parallel Reduction Complexity

- Assume that the number of elements in array is of the form $N=2^D$

- $\log(N)$ parallel stages, each stage $S$ requires $N/2^S$ independent ops
  - Stage Complexity is $O(\log N)$

- For $N=2^D$, approach requires a total of $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is $O(N)$ – It is work-efficient
  - That is, it does not perform more operations than a sequential algorithm

- Time complexity, for $P$ threads physically in parallel ($P$ processors): $O(N/P + \log N)$
  - Compare to $O(N)$ for sequential reduction
  - In a thread block, $N=P$, so $O(\log N)$
What About Cost?

- **Cost** of a parallel algorithm is processors × time complexity
  - Allocate threads instead of processors: $O(N)$ threads
  - Time complexity is $O(\log N)$, so cost is $O(N \log N)$: not cost efficient!

- Brent’s theorem suggests $O(N/\log N)$ threads
  - Each thread does $O(\log N)$ sequential work
  - Then all $O(N/\log N)$ threads cooperate for $O(\log N)$ stages
  - Cost = $O((N/\log N) \times \log N) = O(N) \rightarrow$ cost efficient

- Sometimes called *algorithm cascading*
  - Can lead to significant speedups in practice
Algorithm Cascading

- Combine sequential and parallel reduction
  - Each thread loads and sums multiple elements into shared memory
  - Tree-based reduction in shared memory

- Brent’s theorem says each thread should sum $O(\log n)$ elements
  - i.e. 1024 or 2048 elements per block vs. 256

- Probably beneficial to push it even further
  - Possibly better latency hiding with more work per thread
  - More threads per block reduces levels in tree of recursive kernel invocations
  - High kernel launch overhead in last levels with few blocks

- On G80, best performance with 64-256 blocks of 128 threads
  - 1024-4096 elements per *thread*
Kernel 7, Comments

- For the first six kernels a large number of blocks was used to “tile” the array.

- Kernel 7: reduce the number of blocks and have a thread do more work than just fetch something to shared memory.

- Example [cooked up, not related to actual CUDA warp size, typical CUDA block dim, etc.]:
  - Say you have 1024 elements stored in an array; you need to reduce that array.
  - You start with 32 blocks, each with 4 threads.
  - Then, 128 threads total. It means that a thread, say in block 11, would have to add two numbers, then two numbers, then two numbers, then two more numbers.
  - At this point, everything is in the union of the shared memory associated with the 32 blocks. At this point proceed like before with kernel 6.
Reduction #7: Multiple Adds / Thread

Replace load and add of two elements:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

With a while loop to add as many as necessary:

```c
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
unsigned int gridSize = blockSize*2*gridDim.x;
sdata[tid] = 0;

while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
}
__syncthreads();
```

Note: gridSize loop stride to maintain coalescing!
Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (2^{22} ints)</th>
<th>Bandwidth</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1: interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kernel 2: interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3: sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4: first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5: unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6: completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>Kernel 7: multiple elements per thread</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Kernel 7 on 32M elements: 73 GB/s!
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned int tid) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
    if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
    if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
}

template <unsigned int blockSize>
__global__ void reduce7(int *g_idata, int *g_odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;

    while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize];  i += gridSize;  }
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] += sdata[tid +  64]; } __syncthreads(); }

    if (tid < 32) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
Optimizing CUDA Code
Parallel Prefix Scan Operation in CUDA

October 14, 2013

“A computer will do what you tell it to do, but that may be much different from what you had in mind”
Joseph Weizenbaum
Before We Get Started…

- Last time
  - Tiling as a programming pattern to speed up CUDA code
  - Simple example of finding bugs and improving performance: stencil operation
  - More complex example of optimizing code: vector reduction on the GPU (like your HW)

- Today:
  - Wrap up, vector reduction operation
  - CUDA specific issues that impact GPU computing performance
  - Example: Performing a scan operation

- Miscellaneous
  - CUDA Debugging document uploaded onto the class website:
    - [http://sbel.wisc.edu/Courses/ME964/2013/Lectures/debuggingCUDA.pdf](http://sbel.wisc.edu/Courses/ME964/2013/Lectures/debuggingCUDA.pdf)
  - Exam: Th, November 7, 7:15-9:15 PM (no class on Friday, Nov. 8). Room: 1153ME
    - Review session on Wd, Nov. 6 @ 6 PM in this room (2121ME)
    - Exam will draw on material covered in class and information provided in the primer
    - It'll be a pen and paper exam. Open book and open anything
Vector Reduction, the Journey

- Step 1: we got something running correctly
- Step 2: got rid of modulo operation and eliminated thread divergence
- Step 3: sequential addressing, no more bank conflicts
- Step 4: each threads does an extra first add during load
- Step 5: only one warp active for last part of the algorithm; loop unrolling
- Step 6: templatize code to generate optimal code for any block dimension
- Step 7: have each thread in a block perform several reductions
Performance Comparison

- 1: Interleaved Addressing: Divergent Branches
- 2: Interleaved Addressing: Bank Conflicts
- 3: Sequential Addressing
- 4: First add during global load
- 5: Unroll last warp
- 6: Completely unroll
- 7: Multiple elements per thread (max 64 blocks)

# Elements

131072 262144 524288 1048576 2097152 4194304 8388608 16777216 33554432

Time (ms)
Sources of Efficiency Improvement

- Algorithmic optimizations
  - Changes to addressing, algorithm cascading
  - 11.84x speedup, combined

- Code optimizations
  - Loop unrolling
  - 2.54x speedup, combined
Lessons Learned, Vector Reduction

- Understand CUDA performance characteristics
  - Memory coalescing
  - Warp divergence
  - Bank conflicts
  - Loop unrolling

- Use peak performance metrics to guide optimization (peak bandwidth or peak flop rate)

- Know how to identify type of bottleneck
  - E.g. memory, core computation, or instruction overhead

- Use template parameters to generate optimal code

- Understand parallel algorithm complexity theory (we skipped this part, 7th technique)
CUDA Optimization:
Execution Configuration Heuristics
## Technical Specifications and Features

[Short Detour]

**Legend:**
“multiprocessor” stands for Stream Multiprocessor (what we called SM)

### Compute Capability

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8K</td>
<td>16K</td>
<td>32K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:
“multiprocessor” stands for Stream Multiprocessor (what we called SM)

### Feature Support

(Unlisted features are supported for all compute capabilities)

<table>
<thead>
<tr>
<th>Feature</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory (Section B.11)</td>
<td>No</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Section B.12)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__ballot() (Section B.12)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__threadfence_system() (Section B.5)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td>No</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Blocks per Grid Heuristics

- # of blocks > # of stream multiprocessors (SMs)
  - If this is violated, then you'll have idling SMs

- # of blocks / # SMs > 2
  - Multiple blocks can run concurrently on a multiprocessor
  - Blocks that aren’t waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks waiting to be executed in pipeline fashion
  - To be on the safe side, 1000’s of blocks per grid will scale across multiple generations
  - If you bend backwards to meet this requirement maybe GPU not the right choice
Threads Per Block Heuristics

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough registers to compile and invoke successfully
    - This all depends on your computation, so experiment!

- Use the `nvvp` profiler to understand how many registers you used, what bandwidth you reached, etc.
Occupancy

- In CUDA, executing other warps is the only way to hide latencies and keep the hardware busy

- Occupancy = Number of warps running concurrently on a SM divided by maximum number of warps that can run concurrently
  - When adding up the number of warps, they can belong to different blocks

- Can have up to 48 warps managed by one Fermi SM
  - For 100% occupancy your application should run with 48 warps on an SM

- Many times one can’t get 48 warps going due to hardware constraint
  - See next slide
CUDA Optimization: A Balancing Act

- **Hardware constraints:**
  - Number of registers per kernel
    - 32K per multiprocessor, partitioned among concurrent threads active on the SM
  - Amount of shared memory
    - 16 or 48 KB per multiprocessor, partitioned among SM concurrent blocks

- **Use** `-maxrregcount=N` *flag on nvcc*
  - **N** = desired maximum registers / kernel
  - At some point “spilling” into local memory may occur
    - Might not be that bad, there is L1 cache that helps to some extent

- Recall that you cannot have more than 8 blocks executed by one SM
NVIDIA CUDA Occupancy Calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Varying Block Size

Varying Register Count

Varying Shared Memory Usage
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance
  - If you want to read more about this, there is a Volkov paper on class website
  - What comes to the rescue is the Instruction Level Parallelism (ILP) that becomes an option upon low occupancy

  YET, OFTEN TIMES IS A BAD OMEN …

- Low-occupancy multiprocessors are likely to have a hard time when it comes to hiding latency on memory-bound kernels
  - This latency hiding draws on Thread Level Parallelism (TLP); i.e., having enough threads (warps, that is) that are ready for execution
Parameterize Your Application

- Parameterization helps adaptation to different GPUs

- GPUs vary in many ways
  - # of SMs
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
  - Max. number of warps per SM

- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration
CUDA Instruction Optimization
Need for This Discussion…

- We discussed at length about what you can do to operate at a memory effective bandwidth close to the nominal bandwidth
  - How to access global memory, bank conflict issues in ShMem, etc.

- Next, discuss for 10 minutes about instruction execution throughput

- In the rare situation you have high arithmetic intensity, it’s good to know how fast math gets executed on the device
Instruction Throughput
[How to Evaluate It]

- Throughputs typically given in number of operations per clock cycle per SM
  - Called “nominal throughput” for reasons explained shortly

- Note that for a warp size of 32, one instruction results in 32 operations
  - You have 32 threads operating in lockstep fashion

- Assume that for a given SM, T is the nominal throughput of operations per clock cycle for a certain math instruction
  - Then, 32 operations; i.e., one math instruction, will take x clock cycles, where
    \[ x = \frac{32}{T} \]

- Concluding: instruction throughput – one instruction every \( \frac{32}{T} \) clock cycles
  - In theory, even better: this number gets multiplied by # of SMs in your GPU

- Quick Remark: the higher the T (operations/clock-cycle), the better
## Nominal Throughputs for Native Arithmetic Instructions

[Operations per Clock Cycle per Multiprocessor]

<table>
<thead>
<tr>
<th>Throughput of Native Arithmetic Instructions</th>
<th>Compute Capability 1.x</th>
<th>Compute Capability 2.0</th>
<th>Compute Capability 2.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit floating-point add, multiply, multiply-add</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>64-bit floating-point add, multiply, multiply-add</td>
<td>1</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>32-bit integer add, logical operation</td>
<td>8</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>32-bit integer shift, compare</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit integer multiply, multiply-add, sum of absolute difference</td>
<td>Multiple instructions</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>24-bit integer multiply (__[u]mul24)</td>
<td>8</td>
<td>Multiple instructions</td>
<td>Multiple instructions</td>
</tr>
<tr>
<td>32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base-2 exponential (exp2f), sine (__sinf), cosine (__cosf)</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Type conversions</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
CUDA Instruction Performance

- Instruction performance (per warp), depends on
  - Operand read cycles
  - Nominal instruction throughput
  - Result update cycles

- In other words, instruction performance depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth

- “Cycle” refers to the multiprocessor clock rate
  - 1.4 GHz on the GTX480, for example

This is what we just discussed on the previous two slides
There are two types of runtime math operations

- **__funcf()**: direct mapping to hardware ISA
  - Fast but lower accuracy (see programming guide for details)
  - Examples: __sinf(x), __expf(x), __powf(x,y)

- **funcf()**: compile to multiple instructions
  - Slower but higher accuracy
  - Examples: sinf(x), expf(x), powf(x,y)

- The `-use_fast_math` compiler option forces every funcf() to compile to __funcf()
FP Math is Not Associative!

- In symbolic math, \((x+y)+z == x+(y+z)\)

- This is not necessarily true for floating-point addition

- When you parallelize computations, you likely change the order of operations
  - Round off error propagates differently

- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution

- Beyond this associativity issue, there are many other variables (hardware, compiler, optimization settings) that make sequential and parallel computing results be different
CUDA Optimization: Wrap Up…
Performance Optimization

[Wrapping Up…]

- We discussed many rules and ways to write better CUDA code

- The next several slides sort this collection of recommendations based on their importance

- Writing CUDA software is a craft/skill that is learned
  - Just like playing a game well: know the rules and practice
  - A list of high, medium, and low priority recommendations wraps up discussion on CUDA optimization
    - For more details, check the CUDA C Best Practices Guide:
Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU

4. Strive to have aligned and coalesced global memory accesses

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution)

6. Avoid different execution paths within the same warp
1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts.

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy).

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing.
Use the fast math library whenever speed is very important and you can live with a tiny loss of accuracy.

Prefer faster, more specialized math functions over slower, more general ones when possible.
Writing CUDA Software: Low-Priority Recommendations

1. For kernels with long argument lists, place some arguments into constant memory to save shared memory

2. Use shift operations to avoid expensive division and modulo calculations

3. Avoid automatic conversion of doubles to floats

Example:
Parallel Prefix Scan on the GPU
Software Design Exercise: Parallel Prefix Scan

- Vehicle for software design exercise: parallel implementation of prefix sum
  - Serial implementation – assigned as HW early in the semester
  - Parallel implementation: topic of future assignment

- Goal 1: Flexing our CUDA muscles

- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Different constraints dominate in different applications and/or design solutions

- Goal 3: Identify design patterns that can result in superior parallel performance
  - Understand that there are patterns and it’s worth being aware of them
  - To a large extend, patterns are shaped up by the underlying hardware
Parallel Prefix Sum (Scan)

- **Definition:**
  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity $I$, and an array of $n$ elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the ordered set $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$.

- **Example:**
  If $\oplus$ is addition, then scan on the set $[3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]$ returns the set $[0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]$.

*(From Blelloch, 1990, “Prefix Sums and Their Applications)*
Scan on the CPU

```c
void scan( float* scanned, float* input, int length) {
    scanned[0] = 0;
    for(int i = 1; i < length; ++i) {
        scanned[i] = scanned[i-1] + input[i-1];
    }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
  - Tempted to say that algorithms don’t come more sequential than this…
- Requires exactly \(n-1\) adds
Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential ...
    
    ```
    out[0] = f(0)
    for(j=1; j<n; j++)
        out[j] = out[j-1] + f(j);
    ```

  - ... into parallel:
    ```
    forall(j) in parallel
        temp[j] = f(j);
        scan(out, temp);
    ```

- Useful in implementation of several parallel algorithms:
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.
Parallel Scan Algorithm: Solution #1
Hillis & Steele (1986)

- Note that a implementation of the algorithm shown in picture requires two buffers of length \( n \) (shown is the case \( n=8=2^3 \))
- **Assumption:** the number \( n \) of elements is a power of 2: \( n=2^M \)

<table>
<thead>
<tr>
<th>( d=0 )</th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
<th>( x_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d=1 )</td>
<td>( \Sigma(x_0..x_0) )</td>
<td>( \Sigma(x_0..x_1) )</td>
<td>( \Sigma(x_1..x_2) )</td>
<td>( \Sigma(x_2..x_3) )</td>
<td>( \Sigma(x_3..x_4) )</td>
<td>( \Sigma(x_4..x_5) )</td>
<td>( \Sigma(x_5..x_6) )</td>
<td>( \Sigma(x_6..x_7) )</td>
</tr>
<tr>
<td>( d=2 )</td>
<td>( \Sigma(x_0..x_0) )</td>
<td>( \Sigma(x_0..x_1) )</td>
<td>( \Sigma(x_1..x_2) )</td>
<td>( \Sigma(x_2..x_3) )</td>
<td>( \Sigma(x_3..x_4) )</td>
<td>( \Sigma(x_4..x_5) )</td>
<td>( \Sigma(x_5..x_6) )</td>
<td>( \Sigma(x_6..x_7) )</td>
</tr>
<tr>
<td>( d=3 )</td>
<td>( \Sigma(x_0..x_0) )</td>
<td>( \Sigma(x_0..x_1) )</td>
<td>( \Sigma(x_1..x_2) )</td>
<td>( \Sigma(x_2..x_3) )</td>
<td>( \Sigma(x_3..x_4) )</td>
<td>( \Sigma(x_4..x_5) )</td>
<td>( \Sigma(x_5..x_6) )</td>
<td>( \Sigma(x_6..x_7) )</td>
</tr>
</tbody>
</table>
The Plain English Perspective

- First iteration, I go with stride 1=2^0
  - Start at x[2^M] and apply this stride to all the array elements before x[2^M] to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M - 2^0 additions

- Second iteration, I go with stride 2=2^1
  - Start at x[2^M] and apply this stride to all the array elements before x[2^M] to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M - 2^1 additions

- Third iteration: I go with stride 4=2^2
  - Start at x[2^M] and apply this stride to all the array elements before x[2^M] to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have 2^M - 2^2 additions

- … (and so on)
Consider the $k^{th}$ iteration (where $1 < k < M - 1$): I go with stride $2^{k-1}$

- Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
  - This means that I have $2^M - 2^{k-1}$ additions

...  

$M^{th}$ iteration: I go with stride $2^{M-1}$

- Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
  - This means that I have $2^M - 2^{M-1}$ additions

NOTE: There is no $(M+1)^{th}$ iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of $2^M$ to “&x[2^M] ” it places you right before the beginning of the array – not good…)
Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:

```plaintext
for d := 0 to M-1 do
    forall k in parallel do
        if k - 2^d >= 0 then
            x[out][k] := x[in][k] + x[in][k - 2^d]
        else
            x[out][k] := x[in][k]
    endforall
    swap(in, out)
endfor
```

Double-buffered version of the sum scan
Operation Count
Final Considerations

- The number of operations tally:
  \[(2^M - 2^0) + (2^M - 2^1) + \ldots + (2^M - 2^{k-1}) + \ldots + (2^M - 2^{M-1})\]

  - Final operation count:
    \[M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1\]

  - This is an algorithm with \(O(n \cdot \log(n))\) work

- This scan algorithm is not that work efficient
  - Sequential scan algorithm only needs \(n-1\) additions
  - A factor of \(\log(n)\) might hurt: 20x more work for \(10^6\) elements!
    - Homework requires a scan of about 16 million elements
    - Adding insult to injury: you need two buffers…
```c
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid == 0) ? 0 : g_idata[thid-1];
    __syncthreads();

    for (int offset = 1; offset < n; offset <<= 1) {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
```

Hillis & Steele: Kernel Function
The kernel is very simple, which is good.

Note the pin/pout trick that was used to swap the buffers.

The kernel only works when the entire array is processed by one block:
- One block in CUDA has at most 1024 threads.
- In this setup we cannot handle yet 16 million entries, which is what your assignment will call for.
Parallel Prefix Scan Operation in CUDA
Using streams in CUDA

October 16, 2013

“Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.”
– Norman Augustine

© Dan Negrut, 2013
ME964 UW-Madison
Before We Get Started…

- Last time
  - Wrap up, vector reduction operation
  - CUDA specific issues that impact GPU computing performance
  - Example: Performing a scan operation

- Today:
  - Wrap up: scan operation
  - Streams in CUDA: hiding data movement with useful computation

- Miscellaneous
  - HW 6 posted online. Due on Mo, October 21 at 11:59 pm
  - Start thinking about midterm projects and final projects
    - Due date for midterm project topic selection was Oct 21
    - Moved back to Oct 23 since Oct 21 coincides w/ your due date for HW6
Parallel Scan Algorithm: Solution #1
Hillis & Steele (1986)

- Note that an implementation of the algorithm shown in picture requires two buffers of length \( n \) (shown is the case \( n=8=2^3 \))
- Assumption: the number \( n \) of elements is a power of 2: \( n=2^M \)
Algorithm looks like this:

\[
\text{for } d := 0 \text{ to } M-1 \text{ do } \\
\qquad \text{forall } k \text{ in parallel do } \\
\qquad \qquad \text{if } k - 2^d \geq 0 \text{ then } \\
\qquad \quad x[\text{out}][k] := x[\text{in}][k] + x[\text{in}][k - 2^d] \\
\qquad \text{else} \\
\qquad \quad x[\text{out}][k] := x[\text{in}][k] \\
\qquad \text{endforall} \\
\text{swap}(\text{in}, \text{out}) \\
\text{endfor}
\]

Double-buffered version of the sum scan
The number of operations tally:
\[(2^M-2^0) + (2^M-2^1) + \ldots + (2^M-2^{k-1}) + \ldots + (2^M-2^{M-1})\]

Final operation count:

\[M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1\]

This is an algorithm with \(O(n \cdot \log(n))\) work.

This scan algorithm is not that work efficient

- Sequential scan algorithm only needs \(n-1\) additions
- A factor of \(\log(n)\) might hurt: 20x more work for \(10^6\) elements!
  - Homework requires a scan of about 16 million elements
  - Adding insult to injury: you need two buffers…
```c
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern volatile __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid == 0) ? 0 : g_idata[thid-1];
    __syncthreads();

    for(int offset = 1; offset<n; offset <<= 1) {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] = temp[pin*n+thid] + temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads(); // I need this here before I start next iteration
    }

    g_odata[thid] = temp[pout*n+thid]; // write output
}
```
The kernel is very simple, which is good

Note the pin/pout trick that was used to alternate the destination buffer

The kernel only works when the entire array is processed by one block

- One block in CUDA has at most 1024 threads
- In this setup we cannot handle yet 16 million entries, which is what your assignment will call for
A common parallel algorithm pattern:

**Balanced Trees**

- Build a balanced binary tree on the input data and sweep it to and then from the root
- Tree is not an actual data structure, but a concept to determine what each thread does at each step

For scan:

- Traverse down from leaves to root building partial sums at internal nodes in the tree
  - Root holds sum of all leaves → this is a reduction algorithm
- Traverse back up the tree building the scan from the partial sums
  - Called down-sweep phase

Parallel Scan Algorithm: Solution #2
Harris-Sengupta-Owen (2007)
Picture and Pseudocode ~ Reduction Step~

for k=0 to M-1
  offset = \(2^k\)
  for j=1 to \(2^{M-k-1}\) in parallel do
    \[x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + x[j \cdot 2^{k+1} - 2^k - 1]\]
  endfor
endfor

[Diagram showing the process for k=0 to M-1, with equations and arrays for values of \(d\) from 0 to 3, and operations indicated by arrows and mathematical expressions like \(j \cdot 2^{k+1} - 1\).]

NOTE: “-1” entries indicate no-ops
Operation Count, Reduce Phase

for k=0 to M-1
  offset = $2^k$
  for j=1 to $2^{M-k-1}$ in parallel do
    $x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + x[j \cdot 2^{k+1} - 2^k - 1]$
  endfor
endfor

By inspection:

$$\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$$

Looks promising…
The Down-Sweep Phase

\[
\begin{array}{cccccc}
\times_0 & \Sigma(\times_0..\times_1) & \times_2 & \Sigma(\times_0..\times_3) & \times_4 & \Sigma(\times_4..\times_5) & \times_6 & \Sigma(\times_0..\times_7) \\
\end{array}
\]

\[
\begin{array}{cccccc}
\times_0 & \Sigma(\times_0..\times_1) & \times_2 & \Sigma(\times_0..\times_3) & \times_4 & \Sigma(\times_4..\times_5) & \times_6 & \Sigma(\times_0..\times_3) \\
\end{array}
\]

\[
\begin{array}{cccccc}
\times_0 & 0 & \times_2 & 0 & \times_4 & \Sigma(\times_0..\times_3) & \times_6 & \Sigma(\times_0..\times_5) \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & \times_0 & \Sigma(\times_0..\times_1) & \Sigma(\times_0..\times_2) & \Sigma(\times_0..\times_3) & \Sigma(\times_0..\times_4) & \Sigma(\times_0..\times_5) & \Sigma(\times_0..\times_6) \\
\end{array}
\]

for \( k = M - 1 \) to 0
\[
\text{offset} = 2^k
\]
for \( j = 1 \) to \( 2^{M-k-1} \) in parallel do
\[
\text{dummy} = x[j \cdot 2^{k+1} - 2^k - 1]
\]
\[
x[j \cdot 2^{k+1} - 2^k - 1] = x[j \cdot 2^{k+1} - 1]
\]
\[
x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + \text{dummy}
\]
endfor
endfor

NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…
Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
  - Looks very comparable with the work load in the sequential solution

- The algorithm is convoluted though, it won’t be easy to implement
  - Kernel shown on next slide
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern volatile __shared__ float temp[]; // allocated on invocation
    int thid = threadIdx.x;
    int offset = 1;
    temp[2*thid] = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];
    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            temp[bi] += temp[ai];
        }
        offset <<= 1; // multiply by 2 implemented as bitwise operation
    }
    if (thid == 0) { temp[n - 1] = 0; } // clear the last element
    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();
        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;
            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }
    }
    __syncthreads();
    g_odata[2*thid] = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
Upon first invocation of the kernel (kernel #1), each will bring into shared memory 2048 elements:
- 1024 “lead” elements (see vertical arrows ↑ on slide 9), and…
- 1024 mating elements (the blue, oblique, arrows on slide 9)
- Two consecutive “lead” elements are separated by a stride of \( k=2^1 \)
- A “lead” element and its “mating” element are separated by a stride of \( k/2=1 \)

Suppose you take 6 reduction steps in this first kernel and bail out after writing into the global memory the preliminary data that you computed and stored in shared memory.

The next kernel invocation should pick up the unfinished business where the previous kernel left…
- Call this a “flawless reentry requirement”
Going Beyond 2048 Entries

Upon the second next kernel call, each block will bring into shared memory 2048 elements:

- 1024 “lead” elements, and...
- 1024 “mating” elements
- Two consecutive “lead” elements will now be separated by a stride of $k=2^6$
- A “lead” element and its “mating” element are separated by a stride of $k/2=2^5$
  - Thus, when brining in data from global memory, you are not going to bring over a contiguous chunk of memory of size 2048, rather you’ll have to jump $2^5$ locations between successive “lead and mating element” pairs

- However, once you bring data in shared memory, you process as before
- Before you exit kernel #2 you have to write back data from shared memory into global memory
  - Again, you have to choreograph this shared to global memory store since there is a $2^5$ stride that comes into play
- If you exit kernel #2 after say 4 more reduction steps, the next time you re-enter the kernel (#3) you will have $k=2^{10}$
You will continue the reduction stage until the stride is $2^{M-1}$
- At this point you are ready to start the down-sweep phase
- Down-sweep phase carried out in a similar fashion: we will have to invoke the kernel several times
- Always work in shared memory and copy back data to global memory before bailing out

The challenges here are:
- Understanding the indexing into the global memory to bring data to ShMem
- How to loop across the data in shared memory

There are very many shared memory bank conflicts since you move with strides that are power of 2

Advanced topic: get rid of the bank conflict through padding
Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
- Even if it doesn’t fit like a glove, leads to nice speedup:

<table>
<thead>
<tr>
<th># elements</th>
<th>CPU Scan (ms)</th>
<th>GPU Scan (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.002231</td>
<td>0.079492</td>
<td>0.03</td>
</tr>
<tr>
<td>32768</td>
<td>0.072663</td>
<td>0.106159</td>
<td>0.68</td>
</tr>
<tr>
<td>65536</td>
<td>0.146326</td>
<td>0.137006</td>
<td>1.07</td>
</tr>
<tr>
<td>131072</td>
<td>0.726429</td>
<td>0.200257</td>
<td>3.63</td>
</tr>
<tr>
<td>262144</td>
<td>1.454742</td>
<td>0.326900</td>
<td>4.45</td>
</tr>
<tr>
<td>524288</td>
<td>2.911067</td>
<td>0.624104</td>
<td>4.66</td>
</tr>
<tr>
<td>1048576</td>
<td>5.900097</td>
<td>1.118091</td>
<td>5.28</td>
</tr>
<tr>
<td>2097152</td>
<td>11.848376</td>
<td>2.099666</td>
<td>5.64</td>
</tr>
<tr>
<td>4194304</td>
<td>23.835931</td>
<td>4.062923</td>
<td>5.87</td>
</tr>
<tr>
<td>8388688</td>
<td>47.390906</td>
<td>7.987311</td>
<td>5.93</td>
</tr>
<tr>
<td>16777216</td>
<td>94.794598</td>
<td>15.854781</td>
<td>5.98</td>
</tr>
</tbody>
</table>

Source: 2007 paper of Harris, Sengupta, Owens
Concluding Remarks, Parallel Scan

- The Hillis-Steele (HS) implementation is simple, but suboptimal

- The Harris-Sengupta-Owen (HSO) solution is convoluted, but $O(n)$ scaling
  - The complexity of the algorithm due to an acute bank-conflict situation

- Finally, we have not solved the problem yet: we only looked at the case when our array has up to 1024 elements
  - You will have to think how to handle the $16,777,216 = 2^{24}$ elements case
  - Likewise, it would be fantastic if you implement as well the case when the number of elements is not a power of 2
No penalty if all threads access different banks
  - Or if threads read from the exact same address (multicasting/broadcasting)

This is not the case here: multiple threads access the same shared memory bank with different addresses; i.e. different rows of a bank
  - We have something like $2^{k+1} \cdot j - 1$
    - $k=0$: two way bank conflict
    - $k=1$: four way bank conflict
    - ...

Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions
Initial Bank Conflicts on Load

Each thread loads two shared memory data elements

Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)

```c
temp[2*thid] = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

- Thread 0 accesses banks 0 and 1
- Thread 1 accesses banks 2 and 3
- ...
- Thread 8 accesses banks 16 and 17. Oops, that’s 0 and 1 again…
  - Two way bank conflict, can’t be easily eliminated

Better to load one element from each half of the array

```c
temp[thid] = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

Solution above is helping with the global memory bandwidth as well…
Bank Conflicts in the Tree Algorithm

[Advanced Topics – Supplementary Material]

- When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one.
- We have bank conflicts: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time.

First iteration: 2 threads access each of 8 banks.

Each ✤ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses.
Bank Conflicts in the tree algorithm

[Advanced Topics – Supplementary Material]

- **2nd iteration**: even worse!
  - 4-way bank conflicts; for example:
    - Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.

Bank: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2...

- 3 4 7 4 4 5 6 9 5 13 2 2 3 6 1 10 4 9 7...
- 3 4 7 11 4 5 6 14 5 13 2 15 3 6 1 16 4 9 7...

**2nd iteration**: 4 threads access each of 4 banks.

Each ✰ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses.
Managing Bank Conflicts in the Tree Algorithm
[Advanced Topics – Supplementary Material]

- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks
Use Padding to Reduce Conflicts

After you compute a ShMem address like this:

```
address = 2 * stride * thid;
```

Add padding like this:

```
address += (address >> 4); // divide by NUM_BANKS
```

This removes most bank conflicts
- Not all, in the case of deep trees
  - Material posted online will contain a discussion of this “deep tree” situation along with a proposed solution
Managing Bank Conflicts in the Tree Algorithm

[Advanced Topics – Supplementary Material]

Original scenario.

Bank:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 | 5 | 8 | 2 | 0 | 3 | 1 | 9 | 4 | 5 | 7 |

Modified scenario, virtual 17 bank memory layout.

Virtual Bank:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 |
| 3 | 1 | 7 | 0 | 4 | 1 | 6 | 3 | 5 | 8 | 2 | 0 | 3 | 1 | 9 | P | 4 | 5 | 7 |

Actual physical memory (true bank number)

(0) (1) (2) (3)

Note that only arrows with the same color happen simultaneously.
Bank Conflicts Discussion

[Advanced Topics – Supplementary Material]

- No penalty if all threads access different banks
  - Or if threads read from the exact same address (multicasting/broadcasting)

- This is not the case here: multiple threads access the same shared memory bank with different addresses; i.e. different rows of a bank
  - We have something like $2^{k+1} \cdot j - 1$
    - $k=0$: two way bank conflict
    - $k=1$: four way bank conflict
    - ...

- Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions
Initial Bank Conflicts on Load
[Advanced Topics – Supplementary Material]

- Each thread loads two shared memory data elements

- Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)

  \[
  \text{temp}[2*\text{thid}] = \text{gidata}[2*\text{thid}]; \\
  \text{temp}[2*\text{thid}+1] = \text{gidata}[2*\text{thid}+1];
  \]

  - Thread 0 accesses banks 0 and 1
  - Thread 1 accesses banks 2 and 3
  - ...
  - Thread 8 accesses banks 16 and 17. Oops, that’s 0 and 1 again…
    - Two way bank conflict, can’t be easily eliminated

- Better to load one element from each half of the array

  \[
  \text{temp}[	ext{thid}] = \text{gidata}[	ext{thid}]; \\
  \text{temp}[	ext{thid} + (n/2)] = \text{gidata}[	ext{thid} + (n/2)];
  \]

- Solution above is helping with the global memory bandwidth as well…
Bank Conflicts in the Tree Algorithm

When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one.

We have bank conflicts: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time.

<table>
<thead>
<tr>
<th>Bank:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

First iteration: 2 threads access each of 8 banks.

Each thread corresponds to a single thread. Like-colored arrows represent simultaneous memory accesses.
Bank Conflicts in the tree algorithm

2nd iteration: even worse!

- 4-way bank conflicts; for example:
  Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.

2nd iteration: 4 threads access each of 4 banks.

Each ☐ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses.
Managing Bank Conflicts in the Tree Algorithm
[Advanced Topics – Supplementary Material]

- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks
Use Padding to Reduce Conflicts

After you compute a ShMem address like this:

\[
\text{address} = 2 \times \text{stride} \times \text{tid};
\]

Add padding like this:

\[
\text{address} += (\text{address} >> 4); // \text{divide by NUM_BANKS}
\]

This removes most bank conflicts

- Not all, in the case of deep trees

- Material posted online will contain a discussion of this “deep tree” situation along with a proposed solution
Managing Bank Conflicts in the Tree Algorithm

[Advanced Topics – Supplementary Material]

Original scenario.

Bank:

|   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |   0 | 1 | 2 | ...
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|---|
| 3   | 1 | 7 | 0 | 4 | 1 | 6 | 3 | 5 | 8 | 2  | 0  | 3  | 3  | 1  | 9  | 4  | 5  | 7  |...

Modified scenario, virtual 17 bank memory layout.

Virtual Bank:

|   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |   0 | 1 | 2 | ...
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|---|---|---|---|
| 3   | 1 | 7 | 0 | 4 | 1 | 6 | 3 | 5 | 8 | 2  | 0  | 3  | 3  | 1  | 9  | P  | 4  | 5  | 7  |...

Actual physical memory (true bank number)

(0) (1) (2) (3)

Note that only arrows with the same color happen simultaneously.
CUDA Streams
CUDA Streams: Why Bother?

- In the CPU-GPU interplay, a CUDA enabled GPU can count on two engines
  - An execution engine
  - A copy engine, which actually has 2 subengines that can work simultaneously
    - A H2D copy subengine
    - A D2H copy subengine

- Goal of this segment: learn how to use both engines at the same time

- Remark:
  - In this segment of the lecture the important things happen on the host side, not on the device side
Asynchronous Concurrent Execution

- In order to facilitate concurrent execution on host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task

- Examples of asynchronous calls
  - Kernel launches
  - Device ↔ device memory copies
  - Host ↔ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async

- NOTE: When an application is run via a CUDA debugger or profiler (cuda-gdb, nvvp, Parallel Nsight), all launches are synchronous
Host-Device Data Transfer Issues

- In general, host ↔ device data transfers using `cudaMemcpy()` are blocking
  - Control is returned to the host thread only after the data transfer is complete

- There is a non-blocking variant, `cudaMemcpyAsync()`

```c
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernelll<<<grid,block>>>(a_d);
cpuFunction();
```

- The host does not wait on the device to finish the mem copy and the kernel call for it to start execution of `cpuFunction()` call
- The launch of “kernel” only happens after the mem copy call finishes

- NOTE 1: the asynchronous transfer version requires pinned host memory (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)
- NOTE 2: up until this point we are still not using the two GPU engines
  - We only make the CPU stay busy (which is nonetheless quite good)
Overlapping Host ↔ Device Data Transfer with Device Execution

- When is this overlapping useful?
  - Imagine a kernel executes on the device and only works with the lower half of the device global memory
  - Then, you can copy data from host to device into the upper half of the device global memory
  - These two operations can take place simultaneously

- Note that there is an issue with this idea:
  - The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer

- This issue was addressed by the use of CUDA “streams”
CUDA Streams: Overview

- A programmer can manage concurrency through *streams*

- A stream is a sequence of CUDA commands that execute in issue-order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream
  - NOTE: an operation in a stream does not commence prior to the previous operation being fully completed
    - There is a distinction between queuing an operation in a stream and the moment when it actually starts to be executed on the GPU
CUDA Streams: Overview

- One host thread can define multiple CUDA streams

- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events

- With respect to each other, different CUDA streams execute their commands as they see fit
  - Inter-stream relative behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication for kernels allocated to different streams is undefined)
  - Another way to look at it: streams can be synchronized at barrier points, but correlation of sequence execution within different streams is not supported
CUDA Streams: Creation

- A stream is defined by creating a stream object
  - It is subsequently used by specifying it as the stream parameter to a sequence of kernel launches and host ↔ device memory copies

- The following code sample creates two streams and allocates an array “hostPtr” of float in page-locked memory
  - hostPtr will be used in asynchronous host ↔ device memory transfers

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

- NOTE: As soon you invoke a CUDA function you create a default stream (stream 0)
  - If you don’t explicitly state a stream in the execution configuration of a kernel it is assumed it’s launched as part of stream 0

Notice the length of the array
CUDA Streams: Making Use of Them

- In the code below, each of the two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```c
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we’ll revisit shortly…
CUDA Streams: Clean Up Phase

- Streams are released by calling `cudaStreamDestroy()`
  
  ```c
  for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
  ```

- `cudaStreamDestroy()` waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread
CUDA Streams: Caveats

- Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device ↔ device memory copy,
  - Any CUDA command to stream 0 (including kernel launches and host ↔ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations
CUDA Streams: Synchronization Aspects

`cudaDeviceSynchronize()` halts execution on the host until all preceding commands in all CUDA streams have completed.

`cudaStreamSynchronize()` takes a stream as a parameter and halts execution on the host until all preceding commands in the given CUDA stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

`cudaStreamWaitEvent()` takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()` delay their execution until the given event has completed. Note: this halts the execution of tasks in a stream!

`cudaStreamQuery()` provides applications with a way to know if all preceding commands in a stream have completed.

- **NOTE:** To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.
Example: Use of cudaMemcpyAsync and cudaStreamWaitEvent

- Assume stream1 and stream2 have been defined/initialized already
- The point of this example:
  - Use the two copy subengines at the same time
  - Wait onto the launching of the myKernel until the copy in stream 1 is finished

```c
cudaEvent_t event;
cudaEventCreate (&event); // create event

cudaMemcpyAsync ( d_in, in, size, H2D, stream1 ); // 1) H2D copy of new input
cudaEventRecord ( event, stream1); // record event

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 ); // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event ); // wait for event in stream1
myKernel<<< 1000, 512, 0, stream2 >>> ( d_in, d_out ); // 3) GPU must wait for 1 and 2
someCPUfunction ( blah ) // this gets executed right away
```
How is education supposed to make me feel smarter? Besides, every time I learn something new, it pushes some old stuff out of my brain. Remember when I took that home winemaking course, and I forgot how to drive?"

-- Homer Simpson
Before We Get Started…

- Last time
  - Wrap up: scan operation
  - Started “streams” in CUDA: hiding data movement with useful computation

- Today:
  - Wrap up, Streams in CUDA
  - GPU computing w/ thrust

- Miscellaneous
  - HW due on Mo, October 21 at 11:59 pm
  - Start thinking about midterm projects and final projects
    - Due date for midterm project topic is Oct 23
  - Exam moved back from November 8 to November 25 at 7:15 PM (Room TBA)
    - Review session during regular class hour (show up for review only if you think it’s useful)
Example 1: Using One Stream

- Example draws on material presented in the “CUDA By Example” book
  - J. Sanders and E. Kandrot, authors

- What is the purpose of this example?
  - Shows an example of using page-locked (pinned) host memory

  - Shows one strategy that you should invoke when dealing with applications that require more memory than you can accommodate on the GPU

  - [Most importantly] Shows a strategy that you can follow to get things done on the GPU without blocking the CPU (host) – goes back to the use of cudaMemcpyAsync
    - While the GPU works, the CPU works too

- Remark:
  - In this example the magic happens on the host side. Focus on host code, not on the kernel executed on the GPU (the kernel code is basically irrelevant)
This Example’s Kernel

- Computes some average, it’s not important, simply something that gets done and allows us later on to gauge efficiency gains when using *multiple* streams (for now dealing with one stream only)
  - Inputs: a and b
  - Output: c

```c
#include "../common/book.h"
#define N 1048576 // this is 1024*1024
#define FULL_DATA_SIZE (N*20)
__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```
```c
int main( void ) {
  cudaEvent_t start, stop;
  float elapsedTime;
  cudaStream_t stream;
  int *host_a, *host_b, *host_c;
  int *dev_a, *dev_b, *dev_c;

  // start the timers
  HANDLE_ERROR( cudaEventCreate( &start ) );
  HANDLE_ERROR( cudaEventCreate( &stop ) );

  // initialize the stream; only one stream for now...
  HANDLE_ERROR( cudaStreamCreate( &stream ) );

  // allocate the memory on the GPU
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int)) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int)) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int)) );

  // allocate host pinned memory, used to stream
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );

  for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
  }
```

Stage 1

Stage 2
```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );
    kernel<<<(N+255)/256,256,0,stream>>>( dev_a, dev_b, dev_c );
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream ) );
}
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken:  %3.1f ms\n", elapsedTime );
// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
HANDLE_ERROR( cudaStreamDestroy( stream ) );
return 0;
```
Example 1, Summary

- Stage 1 sets up the events needed to time the execution of the program.
- Stage 2 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device.
- Stage 3 enques the set of GPU operations that need to be undertaken (the “chunkification”).
- Stage 4 needed for timing reporting.
- Stage 5: clean up time.
Example 2: Using Multiple Streams

[Version 2.1]

- Implement the same example but use two streams to this end

- Why would you want to use multiple streams?
  - Overlapping GPU execution with host ↔ device data movement can improve overall performance

- Two ideas underlie the process
  - The idea of “chunkification” of the computation
    - Large computation is broken into pieces that are queued up for execution on the device (we already saw this in Example 1, which uses one stream)
  - The idea of overlapping execution with host ↔ device data movement
  - NOTE: I didn’t want to call this tiling, although it’s similar to that. However, “tiling” is something that happens exclusively on the device (from global to shared memory). Here, the “chunkification” happens on the host
Overlapping Execution and Data Transfer: A Desirable Scenario

Timeline of intended application execution using two independent streams

- **Observations:**
  - “memcpy” actually represents an asynchronous `cudaMemcpyAsync()` memory copy call
  - White (empty) boxes represent time when one stream is waiting to execute an operation that it cannot overlap with the other stream’s operation
  - The goal: keep both GPU engine types (execution and mem copy) busy
    - Note: recent hardware allows two copies to take place simultaneously: one from host to device, at the same time one goes on from device to host (you have two copy subengines)
The “main()” Function, Two Streams

```c
int main( void ) {
  cudaDeviceProp prop;
  int whichDevice;
  HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
  HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
  if (!prop.deviceOverlap) {
    printf( "Device will not handle overlaps, so no speed up from streams\n" );
    return 0;
  }

cudaEvent_t start, stop;
float elapsedTime;
cudaStream_t stream0, stream1;
int *host_a, *host_b, *host_c;
int *dev_a0, *dev_b0, *dev_c0;
int *dev_a1, *dev_b1, *dev_c1;

  // start the timers
  HANDLE_ERROR( cudaEventCreate( &start ) );
  HANDLE_ERROR( cudaEventCreate( &stop ) );

  // initialize the streams
  HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
  HANDLE_ERROR( cudaStreamCreate( &stream1 ) );

  // allocate the memory on the GPU
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c0, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_a1, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_b1, N * sizeof(int) ) );
  HANDLE_ERROR( cudaMalloc( (void**)&dev_c1, N * sizeof(int) ) );

  // allocate host locked memory, used to stream
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
  HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
```

Stage 1

Stage 2

Stage 3
The “main()” Function, Two Streams

[Cntd.]

```c
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

HANDLE_ERROR( cudaEventRecord( start, 0 ) );

// now loop over full data, in bite-sized chunks

for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );

    kernel<<<(N+255),256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

    // copy the data from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );

    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

    // copy the data from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```
The “main()” Function, Two Streams

[Cntd.]

HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );
return 0;
}

NOTE: the kernel doesn’t actually change...
Example 2.1 [Version 1], Summary

- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer.

- Stage 2 sets up the events needed to time the execution of the program.

- Stage 3 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device and initializes data.

- Stage 4 enques the set of GPU operations that need to be undertaken (the “chunkification”).

- Stage 5 takes care of timing reporting and clean up.
Comments, Using Two Streams
[Version 2.1]

- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,”
  - Sanders and Kandrot reported results on NVIDIA GTX285

- Using one stream (in Example 1): 62 ms

- Using two streams (this example, version 1): 61 ms

- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled
The Two Stream Example, Version 2.1
Looking Under the Hood

At the left:
- An illustration of how the work queued up in the streams ends up being assigned by the CUDA driver to the two GPU engines (copy and execution)
- **Important remark:** FIFO is also observed in relation to scheduling the engines (not only the streams)

At the right
- Image shows dependency that is implicitly set up in the two streams given the way the streams were defined in the code
- The queue in the Copy Engine, combined with the implied dependencies determines the scheduling of the Copy and Kernel Engines (see next slide)
The Two Stream Example
Looking Under the Hood

- Note that due to the *specific* way in which the streams were defined (depth first), basically there is no overlap of copy & execution…
  - Explains the no net-gain in efficiency compared to the one stream example

- Remedy: go breadth first, instead of depth first
  - In the current version, execution on the two engines was inadvertently blocked by the way the streams have been set up and the existing scheduling and lack of dependency checks available in the current version of CUDA
The Two Stream Example
[Version 2.2: A More Effective Implementation: Breadth First]

- Old way (the depth first approach):
  - Assign the copy of \( a \), copy of \( b \), kernel execution, and copy of \( c \) to stream0. Subsequently, do the same for stream1

- New way (the breadth first approach):
  - Add the copy of \( a \) to stream0, and then add the copy of \( a \) to stream1
  - Next, add the copy of \( b \) to stream0, and then add the copy of \( b \) to stream1
  - Next, enqueue the kernel invocation in stream0, then enqueue one in stream1.
  - Finally, enqueue the copy of \( c \) back to the host in stream0 followed by the copy of \( c \) in stream1.
The Two Stream Example
A 20% More Effective Implementation (48 vs. 61 ms)

```c
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // enqueue copies of a in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    // enqueue copies of b in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    // enqueue kernels in stream0 and stream1
    kernel<<<(N+255),256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
    kernel<<<(N+255),256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
    // enqueue copies of c from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,   N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```

Copy Engine

```
Stream0: memcpy a
Stream1: memcpy a
Stream0: memcpy b
Stream1: memcpy b
Stream0: memcpy c
Stream1: memcpy c
```

Kernel Engine

```
Stream0: kernel
Stream1: kernel
```

Execution timeline of the breadth-first approach
(blue line shows dependency)

Replaces Previous Stage 4
Using Streams, Lessons Learned

- Streams provide a basic mechanism that enables task-level parallelism in CUDA C applications.

- Two requirements underpin the use of streams in CUDA C:
  - `cudaHostAlloc()` should be used to allocate memory on the host so that it can be used in conjunction with a `cudaMemcpyAsync()` non-blocking copy command.
  - The use of pinned (page-locked) host memory improves data transfer performance even if you only work with one stream.

- Effective latency hiding of kernel execution with memory copy operations requires a breadth-first approach to enqueuing operations in different streams.
  - This is a consequence of the two engine setup associated with a GPU.
Concurrent Kernel Execution

- Fermi: up to 16 kernels can be run on the device at the same time
- When is this useful?
  - Devices of compute capability 2.x are pretty wide (large number of SMs)
  - Sometimes you launch kernels whose execution configuration is smaller than the GPU’s “width”
  - Then, two or three independent kernels can be “squeezed” on the GPU at the same time
- Represents the GPU’s attempt to look like a MIMD architecture
  - Requires use of multiple streams to stand a chance of concurrent kernel execution
GPU Computing using thrust
3 Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach
Maximum Performance

Direction of increased performance (and effort)
Acknowledgments

- The `thrust` slides include material provided by Nathan Bell of NVIDIA

- Slightly modified, assuming responsibility for any mistakes
Design Philosophy, thrust

- Increase programmer productivity
  - Build complex applications quickly

- Encourage generic programming
  - Leverage parallel primitives

- Should run fast
  - Efficient mapping to hardware
What is **thrust**?

- A template library for CUDA
  - Mimics the C++ STL

- Containers
  - On host and device

- Algorithms
  - Sorting, reduction, scan, etc.
What is **thrust**?

[Cntd.]

- **thrust** is a header library – all the functionality is accessed by `#include`-ing the appropriate **thrust** header file

- Program is compiled with **nvcc** as per usual, no special tools are required

- Lots of C++ syntax, related to high-level host-side code that you write
  - The concept of execution configuration, shared memory, etc. : it’s all gone
Why Should One Use thrust?

- Extensively tested
- Open Source
  - Permissive License (Apache v2)
- Active community

NVIDIA [N. Bell]→
Example: Vector Addition

```c
for (int i = 0; i < N; i++)
    Z[i] = X[i] + Y[i];
```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);
    thrust::device_vector<float> Y(3);
    thrust::device_vector<float> Z(3);


    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      thrust::plus<float>());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
Example, Vector Addition

[negrut@euler01 CodeBits]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Jan_12_14:41:45_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221
[negrut@euler01 CodeBits]$ nvcc -O2 exThrust.cu -o exThrust.exe
[negrut@euler01 CodeBits]$ ./exThrust.exe
Z[0] = 25
Z[1] = 55
Z[2] = 40
[negrut@euler01 CodeBits]$  

- Note: file extension should be .cu
Example: SAXPY

```c
for (int i = 0; i < N; i++)
    Z[i] = a * X[i] + Y[i];
```
struct saxpy
{
    float a;

    saxpy(float a) : a(a) {}

    __host__ __device__ float operator()(float x, float y)
    {
        return a * x + y;
    }
};

int main(void)
{
    thrust::device_vector<float> X(3), Y(3), Z(3);


    float aVal = 2.0f;

    thrust::transform(X.begin(), X.end(),
                      Y.begin(),
                      Z.begin(),
                      saxpy(aVal));

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);


    float a = 2.0f;

    thrust::transform(X.begin(), X.end(),
                       Y.begin(),
                       Z.begin(),
                       a * _1 + _2);

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "Z[" << i << "] = " << Z[i] << "\n";

    return 0;
}
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
Containers

- Concise and readable code
  - Avoids common memory management errors
    - e.g.: Vectors automatically release memory when they go out of scope

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Containers

- Compatible with STL containers

```cpp
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method using vector constructor
thrust::device_vector<int> d_vec2(h_list.begin(), h_list.end());
```
Namespaces

- Avoid name collisions

```cpp
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```
Iterators

- A pair of iterators defines a “range”

```c++
// allocate device memory
device_vector<int> d_vec(10);

// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end = d_vec.end();
device_vector<int>::iterator middle = begin + d_vec.size()/2;

// sum first and second halves
int sum_half1 = reduce(begin, middle);
int sum_half2 = reduce(middle, end);

// empty range
int empty = reduce(begin, begin);
```
Iterators

- Iterators act like pointers

```cpp
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```
Iterators

- Encode memory location
  - Automatic algorithm selection

```cpp
// initialize random values on host
host_vector<int> h_vec(100);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```
GPU Computing with **thrust**, wrap up
The CUDA IDE & library ecosystem
October 21, 2013

“The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.”
—Tom Cargill
Before We Get Started...

- Last time
  - Wrap up, Streams in CUDA
  - GPU computing w/ **thrust**
    - New concept: the **functor** as a provider of the “call” operator to customize **thrust** behavior

- Today:
  - Wrap up GPU computing w/ **thrust**
  - Wrap up GPU computing discussion

- Miscellaneous
  - HW due today at 11:59 pm
  - New HW posted online later today. Due on Oct. 28 at 11:59 PM
  - Due date for midterm project topic is Oct 23, 11:59 PM (upload in Learn@UW)
  - Exam moved back from November 8 to November 25 at 7:15 PM (Room TBA)
    - Review session held during regular class hour (show up only if you think it’s useful)
Looking Ahead
[Forum Post]

- Oct. 23, 11:59 PM (Learn@UW submission)
  - Proposal for Midterm Project is due. Default Midterm Project available if undecided.
  - One page of text (doesn’t include title page and references, if any). If going with default, submit one line saying so

- Nov. 15, 11:59 PM (Learn@UW submission)
  - Midterm Project due
  - Seven pages of narrative at the most (this includes pictures but doesn’t include title page and references, if any)

- Nov. 15, 11:59 PM (Learn@UW submission)
  - Proposal for Final Project is due
  - Two pages of text max (this includes pictures but doesn’t include title page and references, if any)

- Nov. 25, 7:15 – 9:15 PM – midterm exam (Room TBA)

- Dec. 15, 11:59 PM (Learn@UW submission)
  - Final Project due
  - Ten pages of narrative at the most (this includes pictures but doesn’t include title page and references, if any)

Use these templates for all documents you submit (stick with this formatting, font size, etc.):
- LaTeX: [http://sbel.wisc.edu/documents/Latextemplate.zip](http://sbel.wisc.edu/documents/Latextemplate.zip)
Algorithms

- Elementwise operations
  - `for_each`, `transform`, `gather`, `scatter` ...

- Reductions
  - `reduce`, `inner_product`, `reduce_by_key` ...

- Prefix Sums [scans]
  - `inclusive_scan`, `inclusive_scan_by_key` ...

- Sorting
  - `sort`, `stable_sort`, `sort_by_key` ...
### Algorithm Description

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>Sum of a sequence</td>
</tr>
<tr>
<td>find</td>
<td>First position of a value in a sequence</td>
</tr>
<tr>
<td>mismatch</td>
<td>First position where two sequences differ</td>
</tr>
<tr>
<td>inner_product</td>
<td>Dot product of two sequences</td>
</tr>
<tr>
<td>equal</td>
<td>Whether two sequences are equal</td>
</tr>
<tr>
<td>min_element</td>
<td>Position of the smallest value</td>
</tr>
<tr>
<td>count</td>
<td>Number of instances of a value</td>
</tr>
<tr>
<td>is_sorted</td>
<td>Whether sequence is in sorted order</td>
</tr>
<tr>
<td>transform_reduce</td>
<td>Sum of transformed sequence</td>
</tr>
</tbody>
</table>
Thrust Example: Sort

```c++
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main() { 
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (805 Mkeys/sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```
Leveraging Parallel Primitives

- Test: sort 32M keys on each platform
  - Performance measured in millions of keys per second [higher is better]
- Conclusion: Use `sort` liberally, it’s highly optimized

<table>
<thead>
<tr>
<th>data type</th>
<th>std::sort</th>
<th>tbb::parallel_sort</th>
<th>thrust::sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>25.1</td>
<td>68.3</td>
<td>3532.2</td>
</tr>
<tr>
<td>short</td>
<td>15.1</td>
<td>46.8</td>
<td>1741.6</td>
</tr>
<tr>
<td>int</td>
<td>10.6</td>
<td>35.1</td>
<td>804.8</td>
</tr>
<tr>
<td>long</td>
<td>10.3</td>
<td>34.5</td>
<td>291.4</td>
</tr>
<tr>
<td>float</td>
<td>8.7</td>
<td>28.4</td>
<td>819.8</td>
</tr>
<tr>
<td>double</td>
<td>8.5</td>
<td>28.2</td>
<td>358.9</td>
</tr>
</tbody>
</table>

Intel Core i7 950 @3.07 GHz

NVIDIA GeForce 480
Input-Sensitive Optimizations

![Sorting Rate vs. Key Bits Diagram]

- Sorting Rate (Mkey/s) vs. Key Bits
  - Key Bits range from 0 to 32
  - Sorting Rate decreases as Key Bits increase

NVIDIA [N. Bell]
Maximum Value

```cpp
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <iostream>

int main(void) {
    thrust::device_vector<float> X(3);

    float init = 0.0f;

    float result = thrust::reduce(X.begin(), X.end(),
                                  init,
                                  thrust::maximum<float>()) ;

    std::cout << "maximum is " << result << "\n";

    return 0;
}
```
Algorithms

- Process one or more ranges

```cpp
// copy values to device
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// sort A in-place
sort(A.begin(), A.end());

// copy A -> B
copy(A.begin(), A.end(), B.begin());

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```
Algorithms

- Standard operators

```cpp
// allocate memory
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```
Algorithms

- Standard data types

```c++
// allocate device memory
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input  = ...;
device_vector<float2> output = ...;

// create function object or ‘functor’
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
// compare x component of two float2 structures
struct compare_float2
{
  __host__ __device__
  bool operator()(float2 a, float2 b)
  {
    return a.x < b.x;
  }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
Custom Types & Operators

```c++
// return true if x is greater than threshold
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...;

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```
Interoperability

- Convert iterators to raw pointers

```c++
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< (N+255) / 256, 256 >>>(N, ptr);

// use ptr in a CUDA API function
cudaMemcpyAsync(ptr, ...);
```
Interoperability

- Wrap raw pointers with `device_ptr`

```cpp
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```
General Transformations

Unary Transformation
\[
\text{for } (\text{int } i = 0; i < N; i++)
\]
\[
X[i] = f(A[i]);
\]

Binary Transformation
\[
\text{for } (\text{int } i = 0; i < N; i++)
\]
\[
X[i] = f(A[i], B[i]);
\]

Ternary Transformation
\[
\text{for } (\text{int } i = 0; i < N; i++)
\]
\[
X[i] = f(A[i], B[i], C[i]);
\]

General Transformation
\[
\text{for } (\text{int } i = 0; i < N; i++)
\]
\[
X[i] = f(A[i], B[i], C[i], ...);
\]

- Like the STL, \texttt{thrust} provides built-in support for unary and binary transformations
- Transformations involving 3 or more input ranges must use a different approach
General Transformations Preamble:

The Zipping Operation

Multiple Distinct Sequences

Unique Sequence of Tuples

zip_iterator
Example: General Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3);
    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(),   Y.end(),   Z.end())),
         U.begin(),
         linear_combo());
    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] << "\n";
    return 0;
}
```

These are the important parts: three different entities are zipped together in one big one

Functor Definition

These are the important parts: three different entities are zipped together in one big one
```cpp
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);


    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
    return 0;
}
thrust, Efficiency Issues
[fusing transformations]
Performance Considerations
[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

![Diagram showing peak flop rate and max bandwidth for Tesla C2050](image-url)

- 1030 GFLOP/s [SinglePrecision]
- 144 GB/s

NVIDIA [N. Bell]→
Arithmetic Intensity
[short detour: 2/3]

Memory bound

Compute bound

SAXPY
FFT
SGEMM

FLOP/Byte
### Arithmetic Intensity

[short detour: 3/3]

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLOP/Byte*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>1 : 12</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2 : 12</td>
</tr>
<tr>
<td>Ternary Transformation</td>
<td>5 : 20</td>
</tr>
<tr>
<td>Sum</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

<table>
<thead>
<tr>
<th>Hardware**</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280</td>
<td>~7.0 : 1</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>~7.6 : 1</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>~6.7 : 1</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>~9.1 : 1</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>~7.1 : 1</td>
</tr>
</tbody>
</table>

** lists the number of flop per byte of data to reach peak Flop/s rate

“Byte” refers to a Global Memory byte
Fusing Transformations

for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);

for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);

Loop Fusion

- One way to look at things...
  - Zipping: reorganizing data for thrust processing
  - Fusing: reorganizing computation for efficient thrust processing
typedef thrust::tuple<float, float> Tuple2;
typedef thrust::tuple<float, float, float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x,y,z) = t;
        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;
        return Tuple2(u,v);
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);


    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
         linear_combo());

    return 0;
}
Since the operation is completely memory bound the expected speedup is $\sim 1.6x (=32/20)$
Fusing Transformations

for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];

Loop Fusion

for (int i = 0; i < N; i++)
    sum += F(X[i]);
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);


    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>())
        ;

    std::cout << "sum of squares is " << result << "\n";
    return 0;
}

Fusing Transformations
Fusing Transformations

Original Implementation

Optimized Implementation

- Try to answer this: how many times will we be able to run faster if we fuse?
typedef thrust::tuple<int, int> Tuple;

struct max_index {
  __host__ __device__
  Tuple operator()(Tuple a, Tuple b) {
    if (thrust::get<0>(a) > thrust::get<0>(b))
      return a;
    else
      return b;
  }
};

int main(void) {
  thrust::device_vector<int> X(3), Y(3);

  X[0] = 10; X[1] = 30; X[2] = 20;  // values
  Y[0] =  0; Y[1] =  1; Y[2] =  2;  // indices

  Tuple init(X[0],Y[0]);

  Tuple result = thrust::reduce
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(),   Y.end())),
      init,
      max_index());

  int value, index;  thrust::tie(value,index) = result;

  std::cout << "maximum value is " << value << " at index " << index << "\n";

  return 0;
}
Maximum Index  [better approach]

typedef thrust::tuple<int,int> Tuple;

struct max_index {
    __host__ __device__
    Tuple operator()(Tuple a, Tuple b) {
        if (thrust::get<0>(a) > thrust::get<0>(b))
            return a;
        else
            return b;
    }
};

int main(void) {
    thrust::device_vector<int>     X(3);
    thrust::counting_iterator<int> Y(0);

    Tuple init(X[0],Y[0]);

    Tuple result = thrust::reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y + X.size())),
         init,
         max_index());

    int value, index;    thrust::tie(value,index) = result;

    std::cout << "maximum value is " << value << " at index " << index << "\n";

    return 0;
}
Maximum Index (Optimized)

Original Implementation

- GPU
- DRAM
- 8 Bytes
- 4 Bytes

Optimized Implementation

- GPU
- DRAM
- 4 Bytes

Try to answer this: how many times will we be able to run faster if we fuse?
Good Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

![Thrust Graph](image)
thrust Wrap-Up

- Significant boost in productivity at the price of small performance penalty
  - No need to know of execution configuration, shared memory, etc.

- Key concepts
  - Functor
  - Fusing operations
  - Zipping data
thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
thrust in “GPU Computing Gems”

PDF available at http://goo.gl/adj9S
Example, **thrust**: 
**Processing Rainfall Data**

Rain situation, end of first day, for a set of five observation stations. Results, summarized over a period of time, reported in the table below.

<table>
<thead>
<tr>
<th>day</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>measuremen</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>

Remarks:
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
Example: Processing Rainfall Data

Given the data above, here’re some questions you might ask:

- Total rainfall at a given site
- Total rainfall between given days
- Total rainfall on each day
- Number of days with any rainfall
struct one_site_measurement
{
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {}

    __host__ __device__
    int operator()(thrust::tuple<int, int> t)
    {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement)
{
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
         one_site_measurement(siteID),
         0,
         thrust::plus<int>())
;
Total Rainfall Between Given Days

```cpp
template <typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day,
                                        const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();

    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
```

For this to fly, you’ll need to include several header files (not all for the code snippet above):

```cpp
#include <thrust/device_vector.h>
#include <thrust/binary_search.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```

NVIDIA [N. Bell]→
Number of Days with Any Rainfall

template <typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day)
{
    return thrust::inner_product(day.begin(), day.end() - 1,
                               day.begin() + 1,
                               0,
                               thrust::plus<int>(),
                               thrust::not_equal_to<int>()) + 1;
}

day [0 = 0 ≠ 1 ≠ 2 ≠ 5 = 5 ≠ 6 = 6 ≠ 7 ≠ 8 ... ]

1+ [0 + 1 + 1 + 1 + 0 + 1 + 0 + 1 + 1 ... ]

NVIDIA [N. Bell]→
Total Rainfall on Each Day

```c++
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement, Vector& day_output, Vector& measurement_output)
{
    size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

day_output.resize(N);
measurement_output.resize(N);

thrust::reduce_by_key(day.begin(), day.end(),
    measurement.begin(),
    day_output.begin(),
    measurement_output.begin());
}
```

day [0 1 2 5 6 7 8 ... ]
measurement [9 5 6 3 8 2 6 5 10 ... ]

day_output [0 1 2 5 6 7 8 ... ]
measurement_output [14 6 3 11 8 5 10 ... ]

NVIDIA [N. Bell]→
3 Ways to Accelerate on GPU

Application

Libraries

Directives

Programming Languages

Easiest Approach → Maximum Performance

Direction of increased performance (and effort)
Directives...
OpenACC

- Seeks to become:
  - A standard for directives-based Parallel Programming
  - Provide portability across hardware platforms and compiler vendors

- Promoted by NVIDIA, Cray, CAPS, PGI
OpenACC Specification

- Hardware agnostic and platform independent (CPU only, different GPUs)

- OpenACC is an open standard for directives based computing

- Announced at SC11 [November 2011]

- Caps, Cray, and PGI to ship OpenACC Compilers beginning Q1 2012

- Very early in the release cycle, you can only download and install a trial version
  - Right now it’s more of an vision…
Host code computes an approximation for π:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793;

    const int n=1000000;
    double h   = 1.0 / (double) n;
    double sum = 0.0;

    for( int i=0; i<=n; i++ )
    {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```
The OpenACC Idea

- Code computes an approximation for $\pi$ [might use multi-core or GPU]

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793238462643;
    const int n = 1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;
    // #pragma acc region for
    for( int i=0; i<=n; i++ ) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;
    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism
OpenACC Target Audience

- OpenACC targets three classes of users:
  - Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  - Users with serial codes looking for portable parallel performance with and without GPUs
  - “Hardcore” GPU programmers with existing CUDA ports
OpenACC Perceived Benefits

- Code easier to maintain
- Helps with legacy code bases
- Portable:
  - Can run same code CPU/GPU
- Very much like OpenMP
- Only small performance loss
  - Cray goal: 90% of CUDA
CUDA: Getting More Info…

- More information on this

- CUDA Tools and Ecosystem
  - Described in detail on NVIDIA Developer Zone
    http://developer.nvidia.com/category/zone/cuda-zone
GPU Computing with CUDA: Wrapping Up

- First question you need to ask: is there a GPU library that I can use?

- In your GPU implementation the code is likely going to be memory bound
  - Move data to GPU and keep it here
  - Understand the GPU memory ecosystem and the costs associated with accessing various memory spaces
  - Algorithms that have higher arithmetic intensity will fare well

- JUST DO IT!
  - Avoid “analysis paralysis”
  - Adopt a “crawl – walk – run” approach
  - Go back and profile/optimize once you have something working
  - To “have something working” debug like a pro (cuda-gdb and cuda-memchk)
Libraries...
CUDA Libraries

- Math, Numerics, Statistics
- Dense & Sparse Linear Algebra
- Algorithms (sort, etc.)
- Image Processing
- Signal Processing
- Finance

- In addition to these widely adopted libraries, several less established ones available in the community

cuBLAS: Dense linear algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Levels 1, 2, and 3 BLAS

- New features in CUDA 4.1:
  - New batched GEMM API provides >4x speedup over MKL
  - Useful for batches of 100+ small matrices from 4x4 to 128x128
  - 5%-10% performance improvement to large GEMMs
Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuSPARSE: Sparse linear algebra routines

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods

- New features in 4.1:
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
2 & -1 \\
4 & -1 \\
5 & 9 & 1 \\
-1 & 8 & 3 \\
\end{bmatrix} \begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix} + \beta \begin{bmatrix}
2 \\
0 \\
1 \\
-1 \\
\end{bmatrix}
\]
Good Speedups Compared to Multi-threaded CPU Execution

Sparse matrix test cases on following slides come from:
1. The University of Florida Sparse Matrix Collection
   http://www.cise.ufl.edu/research/sparse/matrices/
   http://www.nvidia.com/object/nvidia_research_pub_001.html

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuFFT: Multi-dimensional FFTs

- Algorithms based on Cooley-Tukey and Bluestein
- Simple interface, similar to FFTW
- Streamed asynchronous execution
- 1D, 2D, 3D transforms of complex and real data
- Double precision (DP) transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- In-place and out-of-place transforms

\[
F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi \frac{n}{N}}
\]

\[
f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi \frac{n}{N}}
\]
Speedups Compared to Multi-Threaded CPU Execution

![cuFFT Graph]

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
  - Supports several output distributions
  - Statistical test results reported in documentation

- New RNGs in CUDA 4.1:
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG
NPP: NVIDIA Performance Primitives

- Arithmetic, Logic, Conversions, Filters, Statistics, Signal Processing, etc.
- This is where GPU computing shines
- 1,000+ new image primitives in 4.1
Development, Debugging, and Deployment Tools

[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- OpenCL
- OpenGL
- Microsoft DirectX
- Microsoft AMP C/C++
- PGI Accelerator
Debugging Tools

- NVIDIA Parallel Nsight for Visual Studio
- NVIDIA CUDA-MEMCHECK for Linux & Mac
- Allinea DDT with CUDA Distributed Debugging Tool
- NVIDIA CUDA-GDB for Linux & Mac
- TotalView for CUDA for Linux Clusters
Performance Analysis Tools

- NVIDIA Parallel Nsight for Visual Studio
- Vampir Trace Collector
- TAU Performance System
- Performance API Library
- NVIDIA Visual Profiler for Linux & Mac
- Under Development
MPI & CUDA Support

Announced beta at SC2011
Announced pre-release at SC2011
As of OFED 1.5.2
Announced beta at SC2011
Announced pre-release at SC2011

NVIDIA [C. Woolley]→
Cluster Management & Job Scheduling

- **Platform Computing**: LSF, HPC, Cluster Manager
- **Bright Computing**: Bright Cluster Manager
- **Adaptive Computing**: NVML Plugin for GPUs
- **ROCKS+ MOAB**: Univa Grid Engine
- **PBS Works**: PBS Professional
- **Ganglia**: Univa Grid Engine
"In theory, there is no difference between theory and practice. In practice there is."
-- Yogi Berra
Before We Get Started…

- Last time
  - Wrapped up GPU computing w/ thrust
  - Wrapped up GPU computing discussion

- Today:
  - Parallel computing on the CPU
  - Get started with OpenMP for parallel computing on multicore CPUs

- Miscellaneous
  - HW07 posted online
    - Due on Oct. 28 at 11:59 PM
  - Due date for midterm project topic is tonight at 11:59 PM (upload in Learn@UW)
  - Exam moved back from November 8 to November 25 at 7:15 PM (Room TBA)
    - Review session held during regular class hour (show up only if you think it’s useful)
Quick Look at Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Very strong machinery for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides: http://www.realworldtech.com/
Quick Look at Hardware

- Actual layout of the chip

- Schematic of the chip organization
  - LLC: last level cache (L3)
  - Three clocks:
    - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
    - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
    - Ring bus and the shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Quick Look at Hardware

- System on Chip (SoC)
  - So many transistors, you can get creative…
  - The CPU integrates now functionality that used to reside mostly on the north bridge
  - Examples:
    - Voltage regulator
    - Display engine
    - Direct media interface (DMI) controller
    - PCI controller
    - Integrated memory controller (IMC)
  - Functional units to provide these services combine to form the “System Agent”
    - Used to be called the “uncore”
Caches

- **Data:**
  - L1 – 32 KB per core
  - L2 – 512 KB or 1024 KB per core
  - L3 – 8 MB per CPU

- **Instruction:**
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- **Cache is a black hole for transistors**
  - Example: 8 MB of L3 translates into:
    - $8 \times 1024 \times 1024 \times 8 \text{ (bits)} \times 6 \text{ (transistors per bit, SRAM)} = 402 \text{ million transistors out of 1.4 billions}$

- **Caches are *very* important for good performance**
Haswell Microarchitecture
[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details: the primer posted online

[http://www.realworldtech.com]→
Moving from HW to SW
Acknowledgements

- Majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission

- Credit given where due: IOMPP
  - IOMPP stands for “Intel OpenMP Presentation”
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

- **Task parallelism**
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, whistling, and scratching your back
    - The simultaneous breathing of your lungs, beating of your heart, liver function, controlling the swallowing, etc.
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
- Provide an overview of OpenMP in three lectures
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

- Advanced topics
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 64 cores
    - You have a large amount of shared memory, say 128GB

- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
OpenMP: What’s Reasonable to Expect

- If you have 64 cores available to you, it is *highly* unlikely to get a speedup of more than 64 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SPs and hiding memory access latency with warp execution

- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
  - It exists implicitly, under the hood, through ILP support
OpenMP: What Is It?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows

- Standardizes task & loop-level parallelism
- Supports coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience

- Current spec is OpenMP 3.1
  - Released in October 2013
  - [http://www.openmp.org](http://www.openmp.org)
  - More than 300 Pages
Before there was **OpenMP**, a common approach to support parallel programming was by use of **pthreads**
- **“pthread”**: POSIX thread
- **POSIX**: Portable Operating System Interface [for Unix]

**pthreads**
- Available originally under Unix and Linux
- Windows ports are also available some as open source projects

Parallel programming with **pthreads**: relatively cumbersome, prone to mistakes, hard to maintain/scale/expand
- Not envisioned as a mechanism for writing scientific computing software
int main(int argc, char *argv[]) {
    parm     *arg;
    pthread_t *threads;
    pthread_attr_t pthread_custom_attr;

    int n = atoi(argv[1]);

    threads = (pthread_t *) malloc(n * sizeof(*threads));
    pthread_attr_init(&pthread_custom_attr);

    barrier_init(&barrier1); /* setup barrier */
    finals = (double *) malloc(n * sizeof(double)); /* allocate space for final result */

    arg=(parm *)malloc(sizeof(parm)*n);
    for( int i = 0; i < n; i++) { /* Spawn thread */
        arg[i].id = i;
        arg[i].noproc = n;
        pthread_create(&threads[i], &pthread_custom_attr, cpi, (void *)(arg+i));
    }

    for( int i = 0; i < n; i++) /* Synchronize the completion of each thread. */
        pthread_join(threads[i], NULL);

    free(arg);
    return 0;
}
```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include <pthread.h>
#include <netinet/in.h>
#define SOLARIS 1
#define ORIGIN 2
#define OS SOLARIS

typedef struct {
    int id;
    int noproc;
    int dim;
} parm;

typedef struct {
    int cur_count;
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
} barrier_t;

void barrier_init(barrier_t * mybarrier) {
    /* barrier */
    /* must run before spawning the thread */
    pthread_mutexattr_t attr;
    # if (OS==ORIGIN)
        pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
        pthread_mutexattr_setproplceiling(&attr, 0);
        pthread_mutex_init(&mybarrier->barrier_mutex, &attr);
    # elif (OS==SOLARIS)
        pthread_mutex_init(&mybarrier->barrier_mutex, NULL);
    # else
    # error "undefined OS"
    # endif
    mybarrier->cur_count = 0;
}

void barrier(int numprocs, barrier_t * mybarrier) {
    pthread_mutex_lock(&mybarrier->barrier_mutex);
    mybarrier->cur_count++;
    if (mybarrier->cur_count==numprocs) {
        pthread_cond_wait(&mybarrier->barrier_cond, &mybarrier->barrier_mutex);
    } else {
        mybarrier->cur_count=0;
        pthread_cond_broadcast(&mybarrier->barrier_cond);
    }
    pthread_mutex_unlock(&mybarrier->barrier_mutex);
}

void* cpi(void *arg) {
    parm *p = (parm *) arg;
    int myid = p->id;
    int numprocs = p->noproc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    double starttime, endtime;

    if (myid == 0) {
        startwtime = clock();
    }
    barrier(numprocs, &barrier1);
    if (rootn==0)
        finals[myid]=0;
    else {
        h = 1.0 / (double) rootn;
        sum = 0.0;
        for(int i = myid + 1; i <= rootn; i += numprocs) {
            x = h * ((double) i - 0.5);
            sum += f(x);
        }
        mypi = h * sum;
    }
    finals[myid] = mypi;
    barrier(numprocs, &barrier1);
    if (myid == 0){
        pi = 0.0;
        for(int i=0; i < numprocs; i++) pi += finals[i];
        endtime = clock();
        printf("pi is approx %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        printf("wall clock time = %f\n",
            (endtime - startwtime) / CLOCKS_PER_SEC);
    }
    return NULL;
}
```
Looking at the previous example (which is not the best written piece of code, lifted from the web…)

- Code displays platform dependency (not portable)
- Code is cryptic, low level, hard to read and maintain
- Requires busy work: fork and joining threads, etc.
  - Burdens the developer
  - Probably in the way of the compiler as well: rather low chances that the compiler will be able to optimize the implementation

Higher level approach to SMP parallel computing for *scientific applications* was in order
OpenMP Programming Model

- **Master thread** spawns a team of threads as needed
  - Managed transparently on your behalf
  - It still relies on thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check the number of threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_get_num_procs()`
  - Explicit locks
    - `omp_[set|unset]_lock()`
  - And several more...

[OMPI]

→ https://computing.llnl.gov/tutorials/openMP/
A Few Syntax Details to Get Started

- Picking up the API - header file in C, or Fortran 90 module
  ```c
  #include "omp.h"
  use omp_lib
  ```

- Most of the constructs in OpenMP are compiler directives or pragmas
  - For C and C++, the pragmas take the form:
    ```c
    #pragma omp construct [clause [clause]...]```
  - For Fortran, the directives take one of the forms:
    ```fortran
    C$OMP construct [clause [clause]...]  
    !$OMP construct [clause [clause]...]  
    *$OMP construct [clause [clause]...]  
    **$OMP construct [clause [clause]...]```
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles was to have the same code, with no modifications and have it run either on an one core machine, or a multiple core machine.

- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas.

- These hidden directives would be picked up by the compiler only if you instruct it to compile in OpenMP mode.
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code.
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>.

---

Step 1: Go here

Step 2: Select /openmp
OpenMP, Compiling Using the Command Line

- Method depends on compiler

- **GCC:**
  
  ```
  $ gcc -o integrate_omp integrate_omp.c -fopenmp
  ```

- **ICC:**
  
  ```
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- **MSVC (not in the express edition):**
  
  ```
  $ cl /openmp integrate_omp.c
  ```
Enabling OpenMP with CMake

```cmake
# Minimum version of CMake required.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME964-omp)

# Include macros from the SBEL utils library
include(SBELUtils.cmake)

# Example OpenMP program
enable_openmp_support()
add_executable(integrate_omp integrate_omp.cpp)
```

With the template

```cmake
find_package("OpenMP" REQUIRED)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}"
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
```

Without the template

Replaces include(SBELUtils.cmake) and enable_openmp_support() above
OpenMP Odds and Ends…

- Controlling the number of threads
  - The default number of threads that a program uses when it runs is the number of online processors on the machine.
  - For the C Shell: `setenv OMP_NUM_THREADS number`
  - For the Bash Shell: `export OMP_NUM_THREADS=number`

- Timing:

```c
#include <omp.h>
stime = omp_get_wtime();
mylongfunction();
etime = omp_get_wtime();
total=etime-stime;
```
Work Plan

- What is OpenMP?
  - **Parallel regions**
    - Work sharing
    - Data environment
    - Synchronization
- Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to structured blocks
  - Structured block, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are exit() function calls in C/C++

### A structured block

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();

    more: res[id] = do_big_job(id);
    if (not_conv (res[id])) goto more;
}
printf("All done\n");
```

### Not a structured block

```c
if (go_now()) goto more;
#pragma omp parallel
{
    int id = omp_get_thread_num();

    more: res[id] = do_big_job(id);
    if (conv (res[id]) goto done;
    goto more;
}
done: if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
Example: Hello World on my Machine

```c
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel
  {
    int myId = omp_get_thread_num();
    int nThreads = omp_get_num_threads();

    printf("Hello World. I'm thread \%d out of \%d.\n", myId, nThreads);
    for(int i=0; i<2; i++)
      printf("Iter:\%d\n",i);
  }
  printf("GoodBye World\n");
}
```

- Here’s my machine (12 core machine):
  Two Intel Xeon X5650 Westmere 2.66GHz 12MB L3 Cache LGA 1366 95Watts Six-Core Processors
OpenMP: Important Remark

- One of the key tenets of OpenMP is that of data independence across parallel jobs.

- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency.

- Since you place the `omp parallel` directive around some code, it is your responsibility to make sure that data dependency is ruled out.
  - Compilers are not smart enough and sometimes and they can’t identify data dependency between what might look as independent parallel jobs.
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

● Advanced topics
Work Sharing

- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three primary avenues for work sharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct

Each of them automatically divides work among threads.
“omp for” construct

```c
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```c
#pragma omp parallel for
for (int i=0; i< MAX; i++) {
    res[i] = huge();
}
```
The Private Clause

- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that thread 1 stores in x is different than the value that thread 2 stores in the variable x
  - Variables are un-initialized; C++ object is default constructed

```c
void* work(float* c, int N) {
  float x, y;
  int i;
  #pragma omp parallel for private(x,y)
  for(i=0; i<N; i++) {
    x = a[i]; y = b[i];
    c[i] = x + y;
  }
}
```
Example: Parallel Mandelbrot

- Objective: create a parallel version of Mandelbrot using OpenMP work sharing clauses to parallelize the computation of Mandelbrot.
Example: Parallel Mandelbrot
[The Important Function; Includes material from IOMPP]

```c
int Mandelbrot (float z_r[][JMAX], float z_i[][JMAX], float z_color[][JMAX], char gAxis ){
    float xinc = (float)XDELTA/(IMAX-1);
    float yinc = (float)YDELTA/(JMAX-1);

    #pragma omp parallel for private(i,j) schedule(static,8)
    for (int i=0; i<IMAX; i++) {
        for (int j=0; j<JMAX; j++) {
            z_r[i][j] = (float) -1.0*XDELTA/2.0 + xinc * i;
            z_i[i][j] = (float) 1.0*YDELTA/2.0 - yinc * j;
            switch (gAxis) {
                case 'V':
                    z_color[i][j] = CalcMandelbrot(z_r[i][j], z_i[i][j] ) /1.0001;
                    break;
                case 'H':
                    z_color[i][j] = CalcMandelbrot(z_i[i][j], z_r[i][j] ) /1.0001;
                    break;
                default:
                    break;
            }
        }
    }
    return 1;
}
```
The schedule Clause

- The `schedule` clause affects how loop iterations are mapped onto threads

```c
schedule(static [,chunk])
```
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

```c
schedule(dynamic[,chunk])
```
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

```c
schedule(guided[,chunk])
```
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”
schedule Clause Example

```c
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i) ) gPrimesFound++;
    }
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

\[ i = \{ 3, 5, 7, 9, 11, 13, 15, 17 \} \]
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization
- Advanced topics
“A programming language is low level when its programs require attention to the irrelevant.”
-- Allan Perlis
Before We Get Started…

- Last time
  - Parallel computing on the CPU
  - Got started with OpenMP for parallel computing on multicore CPUs

- Today:
  - Continue OpenMP discussion:
    - sections
    - tasks
    - data scoping
    - synchronization

- Miscellaneous
  - Exam on Monday, November 25 at 7:15 PM.
    - Room 1163ME
    - Review session held during regular class hour that Monday
Function Level Parallelism

```c
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```

alice, bob, and cy can be computed in parallel
omp sections

- **#pragma omp sections**
- Must be inside a parallel region
- Precedes a code block containing \( N \) sub-blocks of code that may be executed concurrently by \( N \) threads
- Encompasses each `omp section`, see below

- **#pragma omp section**
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads
#pragma omp parallel sections
{
#pragma omp section
  double a = alice();
#pragma omp section
  double b = bob();
#pragma omp section
  double c = cy();
}

double s = boss(a, b);
printf("%6.2f\n", bigboss(s, c));
Advantage of Parallel Sections

- Independent sections of code can execute concurrently → reduces execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```

The pink and green tasks are executed at no additional time-penalty in the shadow of the purple task.
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 2 procs\n");
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
sections, Example: 2 threads
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while ( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while ( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while ( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
sections, Example: 4 threads
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Tasks
  Data environment
  Synchronization
- Advanced topics
OpenMP Tasks

- **Task** – Most important feature added in the 3.0 version of OpenMP

- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer
Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The OS & runtime decide which of the above
- Tasks are composed of
  - **code** to execute
  - **data** environment
  - **internal control variables (ICV)**

Serial

Parallel
Tasks: What Are They?
[More specifics…]

- **Code to execute**
  - The literal code in your program enclosed by the task directive

- **Data environment**
  - The shared & private data manipulated by the task

- **Internal control variables**
  - Thread scheduling and environment variable type controls

- A task is a specific instance of executable code and its data environment, generated when a thread encounters a **task** construct

- **Two activities: (1) packaging, and (2) execution**
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
using namespace std;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr){
    *itrtr *= 2.;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for(int i=0; i<4;++i)
        for(int j=0; j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);

    for( it = test.begin(); it!= test.end(); it++ ) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        {
            while( it != test.end() ) {
                #pragma omp task private(it)
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }

    for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;
    return 0;
}
Compile like:

```
g++ -o testOMP.exe testOMP.cpp
```
A team of threads is created at the `omp parallel` construct.

A single thread is chosen to execute the while loop—call this thread “L”.

Thread L runs the while loop, creates tasks, and fetches next pointers.

Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it.

Each task runs in its own thread.

All tasks complete at the barrier at the end of the parallel region’s construct.

Each task has its own stack space that will be destroyed when the task is completed.

See an example in a bit.

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p!=listEnd) { //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p) { // block 2
            #pragma omp task private(p)
            process(p);
            p = p->next; // block 3
        }
    }
}
```

How about synchronization issues?
Tasks: Synchronization Issues

- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here
Comments: sections vs. tasks

- **sections** have a “static” attribute: things are mostly settled at compile time

- The **tasks** construct is more recent and more sophisticated
  - They have a “dynamic” attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The run time system generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer

- **NOTE:** It’s the developer responsibility to ensure that different tasks can be executed concurrently
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization

- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- **shared** clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct
Data Scoping – What’s Private

- Not everything is shared...

  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- firstprivate
  - Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct
Data Scoping – The Basic Rule

- When in doubt, explicitly indicate who’s what
  - Data scoping: one of the most common sources of errors in OpenMP
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++)
            {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }
        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++)
            {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    } /* end of sections */

    printf("Thread %d done.\n",tid);
} /* end of parallel section */
A, index, and count are shared by all threads, but temp is local to each thread.
Data Scoping Issue: fib Example

Assume that the parallel region exists outside of fib and that fib and the tasks inside it are in the dynamic extent of a parallel region.

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y;
}
```

Values of the private variables not available outside of tasks
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    {
        x = fib(n-1);
    }
    #pragma omp task
    {
        y = fib(n-2);
    }
    #pragma omp taskwait
    return x+y
}
```

Values of the private variables not available outside of tasks

x is a private variable
y is a private variable

Credit: IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait

    return x+y;
}
```

- n is private in both tasks
- x & y are now shared
- we need both values to compute the sum

The values of the x & y variables will be available outside each task construct – after the taskwait.

Credit: IOMPP
Work Plan

What is OpenMP?
- Parallel regions
- Work sharing
- Data environment
- Synchronization

- Advanced topics
Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the nowait clause
  - The nowait clause is applicable to:
    - for clause
    - single clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{
...;
}
```

```c
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);
```

```c
#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
    #pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location
- Special case of a **critical** section, to be discussed shortly
  - Atomic introduces less overhead than **critical**

```c
index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;

#pragma omp parallel for shared(x, y, index)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Credit: IOMPP
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
Race Condition

- A *race condition* is nondeterministic behavior produced when two or more threads access a shared variable at the same time.

- For example, suppose that `area` is shared and both Thread A and Thread B are executing the statement
  
  ```
  area += 4.0 / (1.0 + x*x);
  ```

Credit: IOMPP
Two Possible Scenarios

Order of thread execution causes non-determinant behavior in a data race

Credit: IOMPP
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
Parallel Computing on Multicore CPUs
October 28, 2013

"The empires of the future are the empires of the mind."
-- Winston Churchill
Before We Get Started...

- Last time: OpenMP
  - sections
  - tasks
  - data scoping
  - synchronization

- Today:
  - reduce operations in OpenMP
  - Closing comments on the OpenMP API

- Miscellaneous
  - Forum etiquette: please be nice to each other
  - HW posted online later today. Due next Mo, at 11:59 PM
  - Midterm Project: If you don’t hear from me, it means it’s ok
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for (int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```c
#include <omp.h>

float RES;
#pragma omp parallel
{
#pragma omp for
   for(int i=0; i<niters; i++) {
      float B = big_job(i);

#pragma omp critical (RES_lock)
     consum(B, RES);
   }
}
```

- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional but highly recommended.

Includes material from IOMPP
OpenMP reduction Clause

```
reduction (op:list)
```

- The variables in `list` will be shared in the enclosing parallel region.

- Here’s what happens inside the parallel or work-sharing construct:
  - A private copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads

- At end of construct, local copies are combined through “op” into a single value

Credit: IOMPP
reduction Example

```
#pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
```

- Local copy of `sum` for each thread engaged in the reduction is private
  - Each local sum initialized to the identity operand associated with the operator that comes into play
    - Here we have “+”, so it’s a zero (0)

- All local copies of `sum` added together and stored in “global” variable
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{4} \frac{4.0}{1+x^2} \, dx = \pi \]

```c
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```
OpenMP Reduction Example: Numerical Integration

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1.0/(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1.0 + x*x);
        }
    }

    double my_pi = sum*step;
    printf("Value of integral is: %f\n", my_pi);
    return 0;
}
```
OpenMP Reduction Example:

Output

```
[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o me759.exe
[negrut@euler24 CodeBits]$ ./me759.exe 100000
Value of integral is: 3.141593
```
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Variable Scoping Aspects

- Consider parallelizing the following code

```c
int main() {
    const int n=20;
    int a[n];
    for( int i=0; i<n; i++ )
        a[i] = i;

    //this is the part that needs to
    //be parallelized
    caller(a, n);

    for( int i=0; i<n; i++ )
        printf("a[%d]=%d\n", i, a[i]);
    return 0;
}

void caller(int *a, int n) {
    int i, j, m=3;
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the “callee” subroutine

- If you run the executable 20 times, you get the same results 20 times
First Attempt to Parallelize

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad…
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 70
    ```
- If you run executable 20 times, you get different results
- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)
  - Fortunately, it’s not used in this code for any subsequent computation

- Q: How would you fix this issue with the static variable?
  - Not necessarily to print the values in increasing order, but to make sure there are no race conditions
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)

---

 четыре

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Parallelized Code

- It looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    
    ```shell
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100
    ```

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely
  - No reason for this behavior to change
Concluding Remarks on the OpenMP API
OpenMP: 30,000 Feet Perspective

- Good momentum behind OpenMP owing to the ubiquity of the multi-core chips
- Shared memory, thread-based parallelism
- Relies on the programmer defining parallel regions
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
  - OpenMP 4.0 – complete specifications released in July 2013
OpenMP
The 30,000 Feet Perspective

- Nomenclature:
  - Multicore Communication API (MCAPI)
  - Multicore Resource-sharing API (MRAPI)
  - Multicore Task Management API (MTAPI)
The OpenMP API

- The OpenMP API is a combination of
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`
The “directives” fall into three categories

- Expression of parallelism (flow control)
  - Example: `#pragma omp parallel for`

- Data sharing among threads (communication)
  - Example: `#pragma omp parallel for private(x,y)`

- Synchronization (coordination or interaction)
  - Example: `#pragma omp barrier`
OpenMP 4.0:
Subset of Run-Time Library OpenMP Routines

1. omp_set_num_threads
2. omp_get_num_threads
3. omp_get_max_threads
4. omp_get_thread_num
5. omp_get_thread_limit
6. omp_get_num_procs
7. omp_in_parallel
8. omp_set_dynamic
9. omp_get_dynamic
10. omp_set_nested
11. omp_get_nested
12. omp_set_schedule
13. omp_get_schedule
14. omp_set_max_active_levels
15. omp_get_max_active_levels
16. omp_get_level
17. omp_get_ancestor_thread_num
18. omp_get_team_size
19. omp_get_active_level
20. omp_init_lock
21. omp_destroy_lock
22. omp_set_lock
23. omp_unset_lock
24. omp_test_lock
25. omp_init_nest_lock
26. omp_destroy_nest_lock
27. omp_set_nest_lock
28. omp_unset_nest_lock
29. omp_test_nest_lock
30. omp_get_wtime
31. omp_get_wtick
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"`

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables

[select env variables]

- **OMP_STACKSIZE**
  - Controls the size of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides a hint to an OpenMP implementation about the desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program
  - Example:
    - `setenv OMP_THREAD_LIMIT 8`
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can implement complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
OpenMP, Some Caveats

- There is a lag between the moment a new specification is released and the time a compiler is capable of handling all of its aspects
  - Intel’s compiler is probably most up to speed

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - C/C++: http://openmp.org/mp-documents/OpenMP-4.0-C.pdf
Multi-Core Computing, Next Decade
The Price of 1 MFlops

- 1 Mflops: 1 million floating point operations per second

- 1961:
  - One would have to combine 17 million IBM-1620 computers to reach 1 Mflops
  - At $64K apiece, when adjusted for inflation this would be $\frac{1}{2}$ the 2013 US national debt

- 2000:
  - About $1,000

- 2013:
  - Less than 20 cents out of the value of a workstation
Feature Length on a Chip: Moore’s Law at Work

- 2013 – 22 nm
- 2015 – 14 nm
- 2017 – 10 nm
- 2019 – 7 nm
- 2021 – 5 nm
- 2023 – ??? (carbon nanotubes?)
What Does This Mean?

- One of two things:
  - We either increase the computational power of the chip since you have more transistors
  - Alternatively, we can keep the number of transistors constant but decrease the size of the chip
Increasing the Number of Transistors: Multicore is Here to Stay

- What does that buy us?

- More computational units

- October 2013:
  - Intel Xeon w/ 12 cores – 3 billion transistors (today’s top of the line)
  - 0.2 Tflops, give or take

- Projecting ahead:
  - 2015: 24 cores
  - 2017: about 50 cores
  - 2019: about 100 cores
  - 2021: about 200 cores
Decreasing the Area of the Chip

- Decreasing the chip size: imagine that we want to pack the power of today’s 12 core chip of on tomorrow’s wafer

- Size of chip – assume a square of length “L”
  - 2013: L is about 20 mm
  - 2015: L ≈ 14 mm
  - 2017: L ≈ 10 mm
  - 2019: L ≈ 7 mm
  - 2021: L ≈ 5 mm → a fifth of an inch fits on your phone
Mechanical Engineering: GPU Computing Example
Mechanical Engineering: OpenMP Example
Mechanical Engineering: MPI Computing Example
End: Parallel Computing w/ OpenMP
Beginning: Parallel Computing w/ MPI
ME759
High Performance Computing for Engineering Applications

Parallel Computing with the Message Passing Interface (MPI)
October 30, 2013

“The best things in life aren’t things.”
-- Art Buchwald, Pulitzer Prize winner
Before We Get Started…

● Last time: OpenMP
  ● reduce operations in OpenMP
  ● Closing comments on the OpenMP API

● Today:
  ● Basic concepts related to computing on clusters of CPUs
  ● Getting started on the Message Passing Interface (MPI) standard

● Miscellaneous
  ● HW posted online, due on Mo. Last hard assignment
  ● Midterm Project: If you don’t hear from me by midnight tonight, it means all is ok
  ● Choose your Final Project time slot - see post
Acknowledgments

- Parts of MPI material covered draws on a set of slides made available by the Irish Centre for High-End Computing (ICHEC) - [www.ichec.ie](http://www.ichec.ie)
  - These slides will contain “ICHEC” at the bottom
  - In turn, the ICHEC material was based on the MPI course developed by Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh

- Individual or institutions are acknowledged at the bottom of the slide, like [A. Jacobs]→
MPI: Textbooks, Further Reading…

- **MPI: A Message-Passing Interface Standard** (1.1, June 12, 1995)
- **MPI-2: Extensions to the Message-Passing Interface** (July 18, 1997)
- **Parallel Programming with MPI**, Peter S. Pacheco, Morgan Kaufmann Publishers, 1997 – very good introduction.
- **Parallel Programming with MPI**, Neil MacDonald, Elspeth Minty, Joel Malard, Tim Harding, Simon Brown, Mario Antonioletti. Training handbook from EPCC
Shared Memory Systems

- Memory resources are shared among processors
  - Typical scenario, on a budget: one node with four CPUs, each with 16 cores

- Relatively easy to program since there is a single unified memory space

- Two issues:
  - Scales poorly with system size due to the need for cache coherence
  - Most often, you need more memory than available on the typical multi-core node

- Example:
  - Symmetric Multi-Processors (SMP)
    - Each processor has equal access to RAM

- Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing

[A. Jacobs]→
Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS

- Information is passed between nodes using the network

- No cache coherence and no need for special cache coherency hardware

- Software development: more difficult to write programs for distributed memory systems since the programmer must keep track of memory usage

- Traditionally, this represents the hardware setup that supports MPI-enabled parallel computing
Overview of Large Multiprocessor Hardware Configurations

- Larger multiprocessors
  - Shared address space
    - Symmetric shared memory (SMP)
    - Distributed shared memory (DSM)
  - Distributed address space
    - Commodity clusters: Beowulf and others
    - Custom cluster
      - Cache coherent: ccNUMA
        - SGI Origin/Altix
      - Noncache coherent: Cray T3E, X1
      - Uniform cluster: IBM BlueGene
      - Constellation cluster of DSMs or SMPs
        - SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Euler

~ Hardware Configurations ~
Hardware Relevant in the Context of MPI

Two Components of Euler that are Important

- **CPU**: AMD Opteron 6274 Interlagos 2.2GHz
  - 16-Core Processor (four CPUs per node → 64 cores/node)
  - 8 x 2MB L2 Cache per CPU
  - 2 x 8MB L3 Cache per CPU
  - Thermal Design Power (TDP): 115W

- **HCA**: 40Gbps Mellanox Infiniband interconnect
  - Bandwidth comparable to PCIe2.0 x16 (~32Gbps), yet the latency is rather poor (~1 microsecond)
  - Ends up being the bottleneck in cluster computing
MPI: The 30,000 Feet Perspective

- The same program is launched for execution independently on a collection of cores

- Each core executes the program

- What differentiates processes is their rank: processes with different ranks do different things (“branching based on the process rank”)
  - Very similar to GPU computing, where one thread did work based on its thread index
The Message-Passing Model

- One starts many process on different cores but on each core the process is spawned by launching the same program
  - Process definition [in ME759]: a program counter and address space

- Message passing enables communication among processes that have separate address spaces

- Interprocess communication typically of
  - Synchronization, followed by…
  - … movement of data from one process’s address space to another’s

- Execution paradigm embraced in MPI: Single Program Multiple Data (SPMD)
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**
  - Processor may run many processes

- **Message-Passing Programming Paradigm**
  - Distributed memory
  - Parallel processors
Our View: A **process** is a **program** performing a task on a **processor**

Each processor/process in a message passing program runs a instance/copy of a **program**:
- Written in a conventional sequential language, e.g., C or Fortran,
- The variables of each sub-program have the same **name** but different **locations** (distributed memory) and different **data**!
- Communicate via special send & receive routines (**message passing**)
A First MPI Program

```c
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    } else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize();
    return 0;
}
```

A First MPI Program

Has to be called first, and once

Has to be called last, and once
Program Output

[negrut@euler04 CodeBits]$ mpiexec -np 8 ./a.out
I am a worker: euler04 (rank=1/7)
I am a worker: euler04 (rank=5/7)
I am a worker: euler04 (rank=6/7)
I am a worker: euler04 (rank=3/7)
I am a worker: euler04 (rank=4/7)
I am the master: euler04
I am a worker: euler04 (rank=2/7)
I am a worker: euler04 (rank=7/7)
[negrut@euler04 CodeBits]$
Why Care about MPI?

- Today, MPI is what enables supercomputers to run at PFlops rates
  - Some of these supercomputers might use GPU acceleration though

- Examples of architectures relying on MPI for HPC:
  - IBM Blue Gene L/P/Q (Argonne National Lab – “Mira”)
  - Cray supercomputers (Oakridge National Lab – “Titan”, also uses K20X GPUs)

- MPI has FORTRAN, C, and C++ bindings – widely used in Scientific Computing
MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- One of the early common implementations: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - OpenMPI: a new kid on the block, joint effort of three or four groups (Los Alamos, Tennessee, Indiana University, Europe)
Where Can We Use Message Passing?

- Message passing can be used wherever it is possible for processes to exchange messages:
  - Distributed memory systems
  - Networks of Workstations
  - Even on shared memory systems
MPI vs. CUDA

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?
  
  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing
  
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems
  
- In large scale computing the future likely to belong to heterogeneous architecture
  
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)
MPI: A Second Example Application

- Example out of Pacheco’s book:
  - “Parallel Programming with MPI”
  - Good book, newer edition available

/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *   Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
MPI: A Second Example Application

#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank;    /* rank of process */
    int p;         /* number of processes */
    int source;    /* rank of sender */
    int dest;      /* rank of receiver */
    int tag = 0;   /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s
", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output

[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
MPI, a Third Example: Approximating \( \pi \)

\[
\int_0^1 \frac{4}{1 + x^2} = 4 \cdot \tan^{-1}(1) = \pi
\]

Numerical Integration: Midpoint rule

\[
\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f((i - 0.5) \cdot h)
\]
MPI, a Third Example: Approximating $\pi$

- Use 4 MPI processes (rank 0 through 3)
- In the picture, $n=13$
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: 1,5,9,13
  - Rank 1: 2,6,10
  - Rank 2: 3,7,11
  - Rank 3: 4,8,12
- Each rank computes the area in its associated sub-intervals
- **MPIReduce** is used to sum the areas computed by each rank yielding final approximation to $\pi$
Code for Approximating $\pi$

```cpp
// MPI_PI.cpp : Defines the entry point for the console application.
//
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
```
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) ";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << endl;
}

MPI_Finalize();
return 0;

“As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications.”

Dave Parnas
Before We Get Started...

- Last time: Started the MPI segment of the course
  - Basic concepts related to computing on clusters of CPUs
  - Getting started on the Message Passing Interface (MPI) standard

- Today:
  - MPI practicalities
  - Point-to-point communication in MPI

- Miscellaneous
  - I provided feedback to all students who uploaded a project proposal
    - Email me if you uploaded a proposal yet haven’t heard from me
  - Choose your Final Project presentation time slot - see post
Code for Approximating $\pi$

```
// MPI_PI.cpp : Defines the entry point for the console application.
//
#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
```
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) ";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << endl;
}
MPI_Finalize();
return 0;
Broadcast
[MPI function used in Example]

- A one-to-many communication.
Collective Communications

- Collective communication routines are higher level routines
- Several processes are involved at a time
- May allow optimized internal implementations, e.g., tree based algorithms
  - Require $O(\log(N))$ time as opposed to $O(N)$ for naïve implementation
Reduction Operations
[MPI function used in Example]

- Combine data from several processes to produce a single result
Barriers

- Used implicitly or explicitly to synchronize processes
MPI, Practicalities
MPI on Euler
[Selecting MPI Distribution]

- What’s available: OpenMPI, MVAPICH, MVAPICH2
- OpenMPI is default on Euler
  - This is the only one we’ll support in ME759

- To load OpenMPI environment variables:
  - (This should have been done automatically)

  ```
  $ module load mpi/gcc/openmpi
  ```
### MPI on Euler: [Compiling MPI Code via Cmake]

```cpp
# Minimum version of CMake required.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME964-mpi)

# Include macros from the SBEL utils library
Include(ParallelUtils.cmake)

# Example MPI program
enable_mpi_support()
add_executable(integrate_mpi integrate_mpi.cpp)
target_link_libraries(integrate_mpi ${MPI_CXX_LIBRARIES})

find_package("MPI" REQUIRED)
list(APPEND CMAKE_C_COMPILE_FLAGS ${MPI_C_COMPILE_FLAGS})
list(APPEND CMAKE_C_LINK_FLAGS ${MPI_C_LINK_FLAGS})
include_directories(${MPI_C_INCLUDE_PATH})
list(APPEND CMAKE_CXX_COMPILE_FLAGS ${MPI_CXX_COMPILE_FLAGS})
list(APPEND CMAKE_CXX_LINK_FLAGS ${MPI_CXX_LINK_FLAGS})
include_directories(${MPI_CXX_INCLUDE_PATH})
```

With the template

Without the template

Replacing include(SBELUtils.cmake) and enable_mpi_support() above.
Most MPI distributions provide wrapper scripts named `mpicc` or `mpicxx`
- Adds in `-L`, `-l`, `-I`, etc. flags for MPI
- Passes any options to your native compiler (`gcc`)
- Very similar to what `nvcc` did for CUDA – it’s a compile driver…

```
$ mpicxx -o integrate_mpi integrate_mpi.cpp
```
Running MPI Code on Euler

\texttt{mpiexec [-np #] [-machinefile file] <program> [<args>]}

- **\texttt{-np #}**: Number of processors. Optional if using a machinefile.
- **\texttt{-machinefile file}**: List of hostnames to use. Inside Torque, this file is at $PBS\_NODEFILE$.
- **Your program and its arguments**

- The machinefile/nodefile is required for multi-node jobs with the version of OpenMPI on Euler.
- \texttt{-np} will be set automatically from the machinefile; can select lower, but not higher.
- See the \texttt{mpiexec} manpage for more options.
Example

euler $ qsub -I -l nodes=8:ppn=4:amd,walltime=5:00
qsub: waiting for job 15246.euler to start
qsub: job 15246.euler ready

euler07 $ cd $PBS_O_WORKDIR
euler07 $ mpiexec -machinefile $PBS_NODEFILE ./integrate_mpi
32 32.121040666358297 in 0.998202s

euler07 $ mpiexec -np 16 -machinefile $PBS_NODEFILE ./integrate_mpi
16 32.121040666359455 in 1.524001s

euler07 $ mpiexec -np 8 -machinefile $PBS_NODEFILE ./integrate_mpi
8 32.121040666359136 in 2.171963s

euler07 $ mpiexec -np 4 -machinefile $PBS_NODEFILE ./integrate_mpi
4 32.121040666360585 in 4.600204s

euler07 $ mpiexec -np 2 -machinefile $PBS_NODEFILE ./integrate_mpi
2 32.121040666366788 in 7.615060s

euler07 $ ./integrate_mpi
1 32.121040666353437 in 15.163330s
Why do I get a compilation error "catastrophic error: #error directive: SEEK_SET is #defined but must not be for the C++ binding of MPI" when I compile C++ application?

- Define the `MPICH_IGNORE_CXX_SEEK` macro at compilation stage to avoid this issue. For instance,
  
  ```
  $ mpicc -DMPICH_IGNORE_CXX_SEEK
  ```

Why?

- There are name-space clashes between `stdio.h` and the MPI C++ binding. MPI standard requires `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` names in the MPI namespace, but `stdio.h` defines them to integer values. To avoid this conflict make sure your application includes the `mpi.h` header file before `stdio.h` or `iostream.h` or undefine `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` names before including `mpi.h`. 
MPI Nuts and Bolts
Goals/Philosophy of MPI

- MPI’s prime goals
  - Provide a message-passing interface for parallel computing
  - Make source-code portability a reality
  - Provide a set of services (building blocks) that increase developer’s productivity

- The philosophy behind MPI:
  - Specify a standard and give vendors the freedom to go about its implementation
  - Standard should be hardware platform & OS agnostic – key for code portability
The Rank, as a Facilitator for Data and Work Distribution

- To communicate together MPI processes need identifiers: \texttt{rank = identifying number}

- Work distribution decisions are based on the \texttt{rank}
  - Helps establish which process works on which data
  - Just like we had thread and block indices in CUDA

\begin{itemize}
  \item myrank=0 data
  \item myrank=1 data
  \item myrank=2 data
  \item \texttt{myrank= (size-1) data}
\end{itemize}

communication network
Message Passing

- Messages are packets of data moving between different processes.
- Necessary information for the message passing system:
  - sending process + receiving process \{ i.e., the two “ranks” \}
  - source \textbf{location} + destination \textbf{location}
  - source \textbf{data type} + destination \textbf{data type}
  - source \textbf{data size} + destination \textbf{buffer size}

![Diagram of message passing network]
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank;     /* rank of process */
    int p;          /* number of processes */
    int source;     /* rank of sender */
    int dest;       /* rank of receiver */
    int tag = 0;    /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
[negrut@euler CodeBits]$ mpiexec -np 8 ./greetingsMPI.exe
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
[negrut@euler CodeBits]$
Communicator  MPI_COMM_WORLD

- All processes of an MPI program are members of the default communicator MPI_COMM_WORLD

- MPI_COMM_WORLD is a predefined handle in mpi.h

- Each process has its own rank in a given communicator:
  - starting with 0
  - ending with (size-1)

- You can define a new communicator in case you find it useful
  - Use MPI_Comm_create call. Example creates the communicator DANS_COMM_WORLD

```c
MPI_Comm_create(MPI_COMM_WORLD, new_group, &DANS_COMM_WORLD);
```
MPI_Comm_create

- Synopsis
  ```c
  int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm);
  ```

- Input Parameters
  - `comm` - communicator (handle)
  - `group` - subset of the family of processes making up the `comm` (handle)

- Output Parameter
  - `comm_out` - new communicator (handle)
Point-to-Point Communication

- Simplest form of message passing

- One process sends a message to another process
  - MPI_Send
  - MPI_Recv

- Sends and receives can be
  - Blocking
  - Non-blocking
  - More on this shortly
Point-to-Point Communication

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator, e.g., DANS_COMM_WORLD
- Processes are identified by their ranks in the communicator
The Data Type

- A message contains a number of elements of some particular data type

- MPI data types:
  - Basic data type
  - Derived data types – more on this later

- Data type *handles* are used to describe the type of the data moved around

Example: message with 5 integers

<p>| 2345 | 654  | 96574 | -12 | 7676 |</p>
<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

**Example:**

```c
2345  654  96574  -12  7676
```

count=5

datatype=MPI_INT

```c
int arr[5]
```
MPI_Send & MPI_Recv: The Eager and Rendezvous Flavors

- If you send small messages, the content of the buffer is sent to the receiving partner immediately
  - Operation happens in “eager mode”

- If you send a large amount of data, the sender function waits for the receiver to post a receive before sending the actual data of the message

- Why this eager-rendezvous dichotomy?
  - Because of the size of the data and the desire to have a safe implementation
  - If you send a small amount of data, the MPI implementation can buffer the content and actually carry out the transaction later on when the receiving process asks for data
    - Can’t play this trick if you attempt to move around a huge chunk of data though
NOTE: Each implementation of MPI has a default value (which might change at run time) beyond which a larger MPI_Send stops acting “eager”
- The MPI standard doesn’t provide specifics
- You don’t know how large is too large…

Does it matter if it’s Eager or Rendezvous?
- In fact it does, sometimes the code can hang – example to come

Remark: In the message-passing paradigm for parallel programming you’ll always have to deal with the fact that the data that you send needs to “live” somewhere during the send-receive transaction
MPI_Send & MPI_Recv: Blocking vs. Non-blocking

- Moving away from the Eager vs. Rendezvous modes → they only concern the MPI_Send and MPI_Recv pair

- Messages can be sent with other vehicles than plain vanilla MPI_Send

- The class of send-receive operations can be classified based on whether they are blocking or non-blocking
  
  - Blocking send: upon return from a send operation, you can modify the content of the buffer in which you stored data to be sent since a copy of the data has been sent
  
  - Non-blocking: the send call returns immediately and there is no guarantee that the data has actually been transmitted upon return from send call
    
    - Take home message: before you modify the content of the buffer you better make sure (through a MPI status call) that the send actually completed
Example: Send & Receive

**Non-blocking Alternative: MPI_Isend**

- If non-blocking, the data “lives” in your buffer – that’s why it’s not safe to change it since you don’t know when transaction was closed
  - This typically realized through a MPI_Isend
    - “I” stands for “immediate”

- **NOTE:** there is another way for providing a buffer region but this alternative is blocking
  - Realized through MPI_Bsend
    - “B” stands for “buffered”
  - The problem here is that *you* need to provide this additional buffer that stages the transfer
    - Interesting question: how large should *that* staging buffer be?
  - Adding another twist to the story: if you keep posting non-blocking sends that are not matched by corresponding “MPI_Recv” operations, you are going to overflow this staging buffer
Example: Send & Receive Blocking Options (several of them)

- The plain vanilla MPI_Send & MPI_Recieve pair is blocking
  - It’s safe to modify the data buffer upon return

- The problem with plain vanilla:
  - 1: when sending large messages, there is no overlap of compute & data movement
    - This is what we strived for when using “streams” in CUDA
  - 2: if not done properly, the processes executing the MPI code can hang

- There are several other flavors of send/receive operations, to be discussed later, that can help with concerns 1 and 2 above
The Mechanics of P2P Communication: Sending a Message

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

- **buf** is the starting point of the message with **count** elements, each described with **datatype**

- **dest** is the rank of the destination process within the communicator **comm**

- **tag** is an additional nonnegative integer piggyback information, additionally transferred with the message
  - The **tag** can be used to distinguish between different messages
  - Rarely used
The Mechanics of P2P Communication: Receiving a Message

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- **buf/count/datatype** describe the receive buffer
- Receiving the message sent by process with rank `source` in `comm`
- Only messages with matching `tag` are received
- Envelope information is returned in the `MPI_Status` object `status`
**MPI_Recv:**
The Need for an MPI_Status Argument

- The **MPI_Status** object returned by the call settles a series of questions:
  - The receive call does not specify the size of an incoming message, but only an upper bound
  - If multiple requests are completed by a single MPI function, a distinct error code may need to be returned for each request
  - The source or tag of a received message may not be known if wildcard values were used in a receive operation
The Mechanics of P2P Communication: Wildcarding

- Receiver can wildcard
  - To receive from any source – source = MPI_ANY_SOURCE
  - To receive from any tag – tag = MPI_ANY_TAG
  - Actual source and tag returned in receiver’s status argument
The Mechanics of P2P Communication: Communication Envelope

- Envelope information is returned from MPI_RECV in status.
- `status.MPI_SOURCE` status. `MPI_TAG` count via `MPI_Get_count()`

```c
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count);
```

For a communication to succeed:

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Message data types must match
- Receiver’s buffer must be large enough
Blocking Type: Communication Modes

- **Send communication modes:**
  - Synchronous send → **MPI_SSEND**
  - Buffered [asynchronous] send → **MPI_BSEND**
  - Standard send → **MPI_SEND**
  - Ready send → **MPI_RSEND**

- **Receiving all modes** → **MPI_RECV**
# Cheat Sheet, Blocking Options

<table>
<thead>
<tr>
<th>Sender modes</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_SSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td>needs application-defined buffer to be declared with MPI_BUFFER_ATTACH</td>
</tr>
<tr>
<td><strong>MPI_BSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronous send</td>
<td>Standard send</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_SEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>May be started <strong>only</strong> if the matching receive is already posted!</td>
<td>avoid, might cause unforeseen problems...</td>
</tr>
<tr>
<td><strong>MPI_RSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when the message (data) has arrived</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_RECV</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ME759
High Performance Computing for Engineering Applications

Parallel Computing with the Message Passing Interface (MPI)
November 4, 2013

“You know that uncertainty you feel today? It never goes away. The question is, do you know how to make uncertainty your friend? “
Before We Get Started…

- Last time:
  - MPI practicalities: compiling and running MPI application on Euler
  - Point-to-point communication in MPI: blocking flavors of send/receive

- Today:
  - Wrap up point-to-point communication in MPI: non-blocking flavors
  - Collective action: barriers, communication, operations

- Miscellaneous
  - HW due tonight at 11:59 PM. Most challenging assignment of ME759
  - New assignment posted later today. Due in one week
    - Has to do with thrust
  - Last regular lecture is on Wd. Fr lecture set aside for Midterm Exam
Midterm & Final Project Partitioning

- If you are happy with your Midterm Project, it can become your Final Project
  - A midterm project report will be due nonetheless to show adequate progress
  - Intermediate report in this case should be a formality

- If not happy w/ your Midterm Project selection, Nov. 15 provides the opportunity to bail out
  - Report should be detailed and follow rules spelled out in forum posting

- For SPH default project: the student[s] w/ the fastest implementation will write a paper with Arman, Dan and another lab member

- Please post related questions on forum

- See syllabus for deadlines
Blocking Type: Communication Modes

- Send communication modes:
  - Synchronous send \(\rightarrow\) MPI_SSEND
  - Buffered [asynchronous] send \(\rightarrow\) MPI_BSEND
  - Standard send \(\rightarrow\) MPI_SEND
  - Ready send \(\rightarrow\) MPI_RSEND

- Receiving all modes \(\rightarrow\) MPI_RECV
# Cheat Sheet, Blocking Options

<table>
<thead>
<tr>
<th>Sender modes</th>
<th>Definition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has started</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_SSEND</strong></td>
<td></td>
<td>needs application-defined buffer to be declared with <strong>MPI_BUFFER_ATTACH</strong></td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless an error occurs), irrespective of receiver</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_BSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classic</td>
<td>Standard send</td>
<td>Rendezvous or eager mode. Decided at run time</td>
</tr>
<tr>
<td><strong>MPI_SEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ready send</td>
<td>Started right away. Will work out <strong>only</strong> if the matching receive is already posted!</td>
<td>Blindly do a send. Avoid, might cause unforeseen problems...</td>
</tr>
<tr>
<td><strong>MPI_RSEND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a the message (data) has arrived</td>
<td></td>
</tr>
<tr>
<td><strong>MPI_RECV</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1) Synchronous Sending in MPI
2) Buffered Sending in MPI

- **Synchronous with MPI_Ssend**
  - In synchronous mode, a send will not complete until a matching receive is posted.
    - The sender has to wait for a receive to be posted
    - No buffering of data
    - Used for ensuring the code is healthy and doesn’t rely on buffering

- **Buffered with MPI_Bsend**
  - Send completes once message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet
3) Standard Sending in MPI
4) Ready Sending in MPI

- Standard with MPI_Send
  - Up to the MPI implementation to decide whether to do rendezvous or eager, for performance reasons
    - NOTE: If it does rendezvous, in fact the behavior is that of MPI_SSend
  - Very commonly used

- Ready with MPI_Rsend
  - Will work correctly only if the matching receive has been posted
  - Can be used to avoid handshake overhead when program is known to meet this condition
  - Rarely used, can cause major problems
Most Important Issue: Deadlocking

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs.
Deadlocking, Another Example

- MPI_Send can respond in eager or rendezvous mode
- Example, on a certain machine running MPICH v1.2.1:

PROCESS 0

...  
MPI_Send()  
MPI_Recv()  
...  

Deadlock  
Data size > 127999 bytes  
Data size < 128000 bytes  

No Deadlock

PROCESS 1

...  
MPI_Send()  
MPI_Recv()  
...
Avoiding Deadlocking

● Easy way to eliminate deadlock is to pair MPI_Ssend and MPI_Recv operations the right way:

PROCESS 0

... MPI_Ssend() MPI_Recv() ...

No Deadlock

PROCESS 1

... MPI_Recv() MPI_Ssend() ...

● Conclusion: understand how the implementation works and what its pitfalls/limitations are
Example

- Always succeeds, even if no buffering is done

```
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Will always deadlock, no matter the buffering mode

```c
if(rank==0)
{
    MPI_Recv(...);
    MPI_Send(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```
Example

- Only succeeds if message is at least one of the transactions is small enough and an “eager” mode is triggered

```c
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Send(...);
    MPI_Recv(...);
}
```
Concluding Remarks, Blocking Options

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode
  - → risks with synchronous send

- Synchronous send (**MPI_SSEND**)  
  - risk of deadlock   
  - risk of serialization  
  - risk of waiting → idle time   
  - high latency / best bandwidth

- Buffered send (**MPI_BSEND**)  
  - low latency / bad bandwidth

- Ready send (**MPI_RSEND**)  
  - use never, except you have a 200% guarantee that Recv is already called in the current version and all future versions of your code
Technicalities, Loose Ends: More on the Buffered Send

- Relies on the existence of a buffer, which is set up through a call
  ```c
  int MPI_Buffer_attach(void* buffer, int size);
  ```

- A bsend is a local operation. It does not depend on the occurrence of a matching receive in order to complete

- If a bsend operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete

- Return from an `MPI_Bsend` does not guarantee the message was sent

- Message may remain in the buffer until a matching receive is posted
Technicalities, Loose Ends: More on the Buffered Send [Cntd.]

- Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space.

- The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient.

- **MPI_Bsend** lowers bandwidth since it requires an extra memory-to-memory copy of the outgoing data.

- The **MPI_Buffer_attach** subroutine provides MPI a buffer in the user's memory. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a process at any time.
Technicalities, Loose Ends: Message Order Preservation

- Rule for messages on the same connection; i.e., same communicator, source, and destination rank:
  - Messages do not overtake each other
  - True even for non-synchronous sends

- If both receives match both messages, then the order is preserved
Read This for Assignment 11

- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)

- Repeat this ping-pong with a loop of length 50

- Add timing calls before and after the loop

- For timing purposes, you might want to use
  
  ```
  double MPI_Wtime();
  ```

- `MPI_Wtime` returns a wall-clock time in seconds

- At process 0, print out the transfer time in seconds
  - Might want to use a log scale
More on Timing
[Useful, for Assignment 11]

```c
int main()
{
    double starttime, endtime;
    starttime = MPI_Wtime();
    .... stuff to be timed ....
    endtime = MPI_Wtime();
    printf("That took %f seconds\n", endtime - starttime);
    return 0;
}
```

- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`
More on Timing
[Useful, for Assignment 11; Cntd.]

- Latency = transfer time for zero length messages
- Bandwidth = message size (in bytes) / transfer time

- Message transfer time and bandwidth change based on the nature of the MPI send operation
  - Standard send (MPI_Send)
  - Synchronous send (MPI_Ssend)
  - Buffered send (MPI_Bsend)
  - Etc.
Non-Blocking Communication
Non-Blocking Communications: Motivation

- Overlap communication with execution (just like w/ CUDA):
  - Initiate non-blocking communication
    - Returns Immediately
    - Routine name starting with MPI...
  - Do some work
    - “latency hiding”
  - Wait for non-blocking communication to complete
Non-blocking Send/Receive

- Syntax

```c
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request);
```

- buf - [in] initial address of send buffer (choice)
- count - [in] number of elements in send buffer (integer)
- datatype - [in] datatype of each send buffer element (handle)
- dest - [in] rank of destination (integer)
- tag - [in] message tag (integer)
- comm - [in] communicator (handle)
- request - [out] communication request (handle)

```c
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
               MPI_Comm comm, MPI_Request *request);
```
The Screenplay: Non-Blocking P2P Communication

- Non-blocking send
  - MPI_Isend(...) doing some other work
  - MPI_Wait(...) waiting until operation locally completed

- Non-blocking receive
  - MPI_Irecv(...) doing some other work
  - MPI_Wait(...) waiting until operation locally completed

/// = waiting until operation locally completed
Non-Blocking Send/Receive

Some Tools of the Trade

- Call returns immediately. Therefore, user must worry whether …
  - Data to be sent is out of the send buffer before trampling on the buffer
  - Data to be received has finished arriving before using the content of the buffer

- Tools that come in handy:
  - For sends and receives in flight
    - `MPI_Wait` – blocking - you go synchronous
    - `MPI_Test` – non-blocking - returns quickly with status information
  - Check for existence of data to receive
    - Blocking: `MPI_Probe`
    - Non-blocking: `MPI_Iprobe`
Waiting for isend/ireceive to Complete

- Waiting on a single send
  ```c
  int MPI_Wait(MPI_Request *request, MPI_Status *status);
  ```

- Waiting on multiple sends (get status of all)
  - Till all complete, as a barrier
    ```c
    int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
    ```
  - Till at least one completes
    ```c
    int MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status);
    ```
  - Helps manage progressive completions
    ```c
    int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount, int *indices, MPI_Status *statuses);
    ```
• Flag true means completed

```c
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

```c
int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);
```

```c
int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag,
                 MPI_Status *status);
```

• Like a non blocking MPI_Waitsome

```c
int MPITestsome(int incount, MPI_Request *requests, int *outcount, int *indices,
                 MPI_Status *statuses);
```
The Need for MPI_Probe and MPI_Iprobe

- The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them.

- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status).

- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message.
Probes yield incoming size

- Blocking Probe, wait till match
  ```c
  int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
  ```

- Non Blocking Probe, flag true if ready
  ```c
  int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
  ```
Two types of communication:
- Blocking:
  - Safe to change content of buffer holding on to data in the MPI send call
- Non-blocking:
  - Be careful with the data in the buffer, since you might step on/use it too soon

MPI provides four modes for these two types
- standard, synchronous, buffered, ready
Collective Actions
Collective Actions

- MPI actions involving a group of processes
- Must be called by all processes in a communicator
- All collective actions are blocking

Types of Collective Actions (three of them):
- Global Synchronization (barrier synchronization)
- Global Communication (broadcast, scatter, gather, etc.)
- Global Operations (sum, global maximum, etc.)
Barrier Synchronization

- Syntax:

  ```c
  int MPI_Barrier(MPI_Comm comm);
  ```

- **MPI_Barrier** not needed that often:
  - All synchronization is done automatically by the data communication
    - A process cannot continue before it has the data that it needs
  - If used for debugging
    - Remember to remove for production release
Communication Action: Broadcast

- Function prototype:

```c
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

- rank of the sending process (i.e., root process)
- must be given identically by all processes
MPI_Bcast

A₀ : any chunk of contiguous data described with MPI_Datatype and count
### MPI_Bcast

```c
int MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
```

- **INOUT** : `buffer` (starting address, as usual)
- **IN** : `count` (number of entries in buffer)
- **IN** : `type` (can be user-defined)
- **IN** : `root` (rank of broadcast root)
- **IN** : `com` (communicator)

- Broadcasts message from `root` to all processes (including `root`)
- `com` and `root` must be identical on all processes
- On return, contents of `buffer` is copied to all processes in `com`
Example: MPI_Bcast

- Read a parameter file on a single processor and send data to all processes

```c
#include "mpi.h"
#include <assert.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs;
    float data = -1.0;
    FILE *file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    if( myRank==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file != NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);
    MPI_Finalize();
}
```
Example: MPI_Bcast

[Output]

[negrut@euler CodeBits]$ qsub -I -l nodes=8:ppn=4,walltime=5:00
qsub: waiting for job 16114.euler to start
qsub: job 16114.euler ready

[negrut@euler17 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler17 CodeBits]$ mpiexec -np 4 a.out
data before: -1.000000
data before: -1.000000
data before: -1.000000
data before: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
Communication Action: Gather

- Function Prototype

```c
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                   int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

e.g., root=1
MPI_Gather

数据 (缓冲区)

[A. Siegel]→
MPI_Gather

```c
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** `sendbuf`  (starting address of send buffer)
- **IN** `sendcount`  (number of elements in send buffer)
- **IN** `sendtype`  (type)
- **OUT** `recvbuf`  (address of receive buffer)
- **IN** `recvcount`  (n-elements for any single receive)
- **IN** `recvtype`  (data type of recv buffer elements)
- **IN** `root`  (rank of receiving process)
- **IN** `comm`  (communicator)
MPI_Gather

- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- Remarks:
  - Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
  - recvcount on root indicates number of items received from each process, not total. This is a very common error
- Exercise: Sketch an implementation of MPI_Gather using only send and receive operations.
```c
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = nlcl */
    data_loc = (float *) malloc(nlcl*sizeof(float));

    for (i = 0; i < nlcl; ++i) data_loc[i] = myRank;

    if (myRank == 0)  data = (float *) malloc(nprocs*sizeof(float)*nlcl);

    MPI_Gather(data_loc, nlcl, MPI_FLOAT, data, nlcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0){
        for (i = 0; i < nlcl*nprocs; ++i){
            printf("%f\n", data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
Communication Action: Scatter

- Function prototype

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Example:
- `root=1`

Diagram:
- Before scatter:
  - A
  - B
  - C
  - D
  - E
- After scatter:
  - A
  - B
  - C
  - D
  - E
  - ABODE

Note:
- The diagram illustrates the communication action of Scatter with an example where root=1.
MPI_Scatter

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scatter

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₅</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[A. Siegel]→
MPI_Scatter

```c
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- **IN** sendbuf  (starting address of send buffer)
- **IN** sendcount  (number of elements sent to each process)
- **IN** sendtype  (type)
- **OUT** recvbuf  (address of receive buffer)
- **IN** recvcount  (n-elements in receive buffer)
- **IN** recvtype  (data type of receive elements)
- **IN** root  (rank of sending process)
- **IN** comm  (communicator)
MPI_Scatter

- Inverse of MPI_Gather

- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter

- Remarks:
  - All arguments are significant on root, while on other processes only recvbuf, recvcount, recvtype, root, and comm are significant
```c
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl,
                MPI_FLOAT, 0, MPI_COMM_WORLD);

    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ){
            for (int j = 0; j < n_lcl; ++j) printf("%f\n", data_l[j]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

This is interesting. Think what's happening here...
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$

Putting Things in Perspective...

- Gather: you automatically create a serial array from a distributed one

- Scatter: you automatically create a distributed array from a serial one
Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- \( d_0 \circ d_1 \circ d_2 \circ d_3 \circ \ldots \circ d_{s-2} \circ d_{s-1} \)
  - \( d_i \) = data in process rank \( i \)
    - single variable, or
    - vector
  - \( \circ \) = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation

- Floating point rounding may depend on usage of associative law:
  - \([(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ […] \circ (d_{s-2} \circ d_{s-1})\]
  - \((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ […] ) \circ d_{s-2}) \circ d_{s-1})\)
Example of Global Reduction

- Global integer sum
- Sum of all `inbuf` values should be returned in `resultbuf`.
- Assume root=0;

```c
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD);
```

- The result is only placed in `resultbuf` at the root process.
### Predefined Reduction Operation Handles

<table>
<thead>
<tr>
<th>Predefined operation handle</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of the maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of the minimum</td>
</tr>
</tbody>
</table>
MPI_Reduce

before MPI_REDUCE

- inbuf
- result

A B C
D E F
G H I
J K L
M N O

after

A B C
D E F
G H I
J K L
M N O

root=1

AoDoGoJoM

[ICHEC]→
ME759
High Performance Computing for Engineering Applications

Parallel Computing with the Message Passing Interface (MPI)
November 6, 2013

"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."
-- Winston Churchill
Before We Get Started…

- Last time:
  - Wrap up point-to-point communication in MPI: non-blocking flavors
  - Collective action: barriers, communication, operations

- Today:
  - Collective action: operations
  - User defined types in MPI
  - Departing thoughts: CUDA, OpenMP, MPI

- Miscellaneous
  - No class on Friday. Time slot set aside for midterm exam
  - Midterm exam is Nov. 25 at 7:15 PM in room 1163ME
    - Review session on Monday, Nov 25 during regular class. Attend if you have questions
  - I will travel and miss four office hours: next week and subsequent week
    - I am checking my email on daily basis
  - Final Project Proposal due at 11:59 PM on Nov. 15
If you are happy with your Midterm Project, it can become your Final Project
- No midterm project report due then

If not happy w/ your Midterm Project selection: November 15 provides the opportunity to wrap up and choose a different Final Project
- Report should be detailed and follow rules spelled out in forum posting

Nov 15: Final Project proposal should be uploaded
- Do so even if you choose to continue Midterm Project
  - In this case simply upload a one liner stating this
- If changing to new project, submit a proposal that details the work to be done

For SPH default project: the student[s] w/ the fastest implementation will write a paper with Arman, Dan and another lab member
MPI_Reduce

Before MPI_REDUCE:
- inbuf
- result

After:
- root=1
- AoDoGoJoM

[Diagram showing the reduction process]
Reduce Operation

Assumption: Rank 0 is the root
MPI_Reduce

```c
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `root` (rank of root process)
- **IN** `comm` (communicator)
MPI_Reduce example

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, 6, \text{MPI\_INT}, \text{MPI\_SUM}, 0, \text{MPI\_COMM\_WORLD})
\]
MPI_Reduce, MPI_Allreduce

- **MPI_Reduce**: result is collected by the root only
  - The operation is applied element-wise for each element of the input arrays on each processor

- **MPI_Allreduce**: result is sent out to everyone

```c
... MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD)
...```

```c
... MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD)
...```

Credit: Allan Snively
MPI_Allreduce

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allreduce

<table>
<thead>
<tr>
<th></th>
<th>A0+A1+A2</th>
<th>B0+B1+B2</th>
<th>C0+C1+C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MPI_Allreduce

```c
int MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `comm` (communicator)
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int my_rank, nprocs, gsum, gmax, gmin, data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    data_l = my_rank;

    MPI_Allreduce(&data_l, &gsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmax, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    MPI_Allreduce(&data_l, &gmin, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);

    printf("gsum: %d, gmax: %d  gmin:%d\n", gsum,gmax,gmin);
    MPI_Finalize();
}
```
Example: MPI_Allreduce

[Output]

[negrut@euler24 CodeBits]$ mpiexec -np 10 me759.exe
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
gsum: 45, gmax: 9 gmin:0
[negrut@euler24 CodeBits]$
MPI_SCAN

- Performs a prefix reduction on data distributed across a communicator

- The operation returns, in the receive buffer of the process with rank $i$, the reduction of the values in the send buffers of processes with ranks $0, \ldots, i$ (inclusive)

- The type of operations supported, their semantics, and the constraints on send and receive buffers are as for MPI_REDUCE
MPI_SCAN

before MPI_SCAN

- inbuf
- result

after

A  A o D  A o D o G  A o D o G o J  A o D o G o J o M

done in parallel
Scan Operation

**Processes**

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

**Data (input buffer)**

**Data (output buffer)**

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1</td>
<td>B0+B1</td>
<td>C0+C1</td>
</tr>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
MPI_Scan: Prefix reduction

- Process $i$ receives data reduced on process 0 through $i$

\[
\text{MPI}_\text{Scan}(\text{sbuf}, \text{rbuf}, 6, \text{MPI_INT}, \text{MPI}_\text{SUM}, \text{MPI}_\text{COMM}_\text{WORLD})
\]
MPI_Scan

```c
int MPI_Scan (void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `comm` (communicator)

- **Note:** `count` refers to total number of elements that will be received into receive buffer after operation is complete.
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray*sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n) {
        if( myRank == n ) {
            for(i=0; i<dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray*sizeof(int));
    MPI_Scan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    free(result); free(data_l);
    return 0;
}
Example: MPI_Scan

[Output]

```
[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.exe
Process 0. Entry: 0.  Value: 0
Process 0. Entry: 1.  Value: 0

Process 1. Entry: 0.  Value: 1
Process 1. Entry: 1.  Value: 2

Process 2. Entry: 0.  Value: 2
Process 2. Entry: 1.  Value: 4

Process 3. Entry: 0.  Value: 3
```

```
Post Scan – Content on Process: 0
Entry: 0.  Scan Val: 0
Entry: 1.  Scan Val: 0

Post Scan – Content on Process: 1
Entry: 0.  Scan Val: 1
Entry: 1.  Scan Val: 2

Post Scan – Content on Process: 2
Entry: 0.  Scan Val: 3
Entry: 1.  Scan Val: 6

Post Scan – Content on Process: 3
Entry: 0.  Scan Val: 6
Entry: 1.  Scan Val: 12
```

[negrut@euler26 CodeBits]$
```
MPI_Exscan

- **MPI_Exscan** is like **MPI_Scan**, except that the contribution from the calling process is not included in the result at the calling process (it is contributed to the subsequent processes).

- The value in `recvbuf` on the process with rank 0 is undefined, and `recvbuf` is not significant on process 0.

- The value in `recvbuf` on the process with rank 1 is defined as the value in `sendbuf` on the process with rank 0.

- For processes with rank \( i > 1 \), the operation returns, in the receive buffer of the process with rank \( i \), the reduction of the values in the send buffers of processes with ranks \( 0, \ldots, i-1 \) (inclusive).

- The type of operations supported, their semantics, and the constraints on send and receive buffers, are as for **MPI_REDUCE**.
**MPI_Exscan**

```c
int MPI_Exscan (void  *sendbuf, void  *recvbuf, int  count,
                MPI_Datatype  datatype, MPI_Op  op, MPI_Comm  comm);
```

- **IN** `sendbuf` (address of send buffer)
- **OUT** `recvbuf` (address of receive buffer)
- **IN** `count` (number of elements in send buffer)
- **IN** `datatype` (data type of elements in send buffer)
- **IN** `op` (reduce operation)
- **IN** `comm` (communicator)
```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, i, n;
    int *result, *data_l;
    const int dimArray = 2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    data_l = (int *) malloc(dimArray * sizeof(int));
    for (i = 0; i < dimArray; ++i) data_l[i] = (i+1)*myRank;
    for (n = 0; n < nprocs; ++n){
        if( myRank == n ){
            for (i = 0; i < dimArray; ++i) printf("Process %d. Entry: %d. Value: %d\n", myRank, i, data_l[i]);
            printf("\n");
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    result = (int *) malloc(dimArray * sizeof(int));
    MPI_Exscan(data_l, result, dimArray, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (n = 0; n < nprocs; ++n){
        if (myRank == n) {
            printf("\n Post Scan - Content on Process: %d\n", myRank);
            for (i = 0; i < dimArray; ++i) printf("Entry: %d. Scan Val: %d\n", i, result[i]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```
Example: MPI_Exscan

[Output]

[negrut@euler26 CodeBits]$ mpicxx -o me759.exe testMPI.cpp
[negrut@euler26 CodeBits]$ mpiexec -np 4 me759.exe

Process 0. Entry: 0. Value: 0
Process 0. Entry: 1. Value: 0

Process 1. Entry: 0. Value: 1
Process 1. Entry: 1. Value: 2

Process 2. Entry: 0. Value: 2
Process 2. Entry: 1. Value: 4

Process 3. Entry: 0. Value: 3

Post Scan – Content on Process: 0
Entry: 0. Scan Val: 321045752
Entry: 1. Scan Val: 32593

Post Scan – Content on Process: 1
Entry: 0. Scan Val: 0
Entry: 1. Scan Val: 0

Post Scan – Content on Process: 2
Entry: 0. Scan Val: 1
Entry: 1. Scan Val: 2

Post Scan – Content on Process: 3
Entry: 0. Scan Val: 3
Entry: 1. Scan Val: 6

[negrut@euler26 CodeBits]$
User-Defined Reduction Operations

- Operator handles
  - Predefined – see table of last lecture: MPI_SUM, MPI_MAX, etc.
  - User-defined

- User-defined operation ■:
  - Should be associative
  - User-defined function must perform the operation “vector_A ■ vector_B”

- Registering a user-defined reduction function:

  ```c
  MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
  ```

- commute tells the MPI library whether func is commutative or not
Example: Norm 1 of a Vector

```c
#include <mpi.h>
#include <stdio>
#include <math.h>

void oneNorm(float *in, float *inout, int *len, MPI_Datatype *type) {
    int i;
    for (i=0; i<*len; i++) {
        *inout = fabs(*in) + fabs(*inout);        /* one-norm */
        in++; inout++;
    }
}

int main(int argc, char* argv[]) {
    int root=0, p, myid;
    float sendbuf, recvbuf;
    MPI_Op myop;

    int commutes=1;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    //create the operator...
    MPI_Op_create(onenorm, commune, &myop);

    //get some fake data used to make the point...
    sendbuf = myid*(-1)^myid;
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Reduce (&sendbuf, &recvbuf, 1, MPI_FLOAT, myop, root, MPI_COMM_WORLD);
    if( myid == root )
        printf("The operation yields %f
", recvbuf);
    MPI_Finalize();
    return 0;
}
```
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <cmath>

template <typename T> struct absval {
    __host__ __device__
    T operator()(const T& x) const {
        return fabs(x);
    }
};

int main(void) {
    // initialize host array
    float x[4] = {1.0, -2.0, 3.0, -4.0};

    // transfer to device
    thrust::device_vector<float> d_x(x, x + 4);

    absval<float> unary_op;
    float res = thrust::transform_reduce(d_x.begin(), d_x.end(), unary_op, 0.f, thrust::plus<float>());

    std::cout << res << std::endl;
    return 0;
}
MPI Derived Types
[Describing Non-contiguous and Heterogeneous Data]
The Relevant Question

- The relevant question that we want to be able to answer?
  - “What’s in your buffer?”

- Communication mechanisms discussed so far allow send/recv of a contiguous buffer of identical elements of predefined data types

- Often want to send non-homogenous elements (structure) or chunks that are not contiguous in memory

- MPI enables you to define derived data types to answer the question “What’s in your buffer?”
MPI Datatypes

- MPI Primitive Datatypes
  - MPI_Int, MPI_Float, MPI_INTEGER, etc.

- Derived Data types - can be constructed by four methods:
  - contiguous
  - vector
  - indexed
  - struct

  Can be subsequently used in all point-to-point and collective communication

- The motivation: create your own types to suit your needs
  - More convenient
  - More efficient
Type Maps

A derived data type specifies two things:
- A sequence of primitive data types
- A sequence of integers that represent the byte displacements, measured from the beginning of the buffer

Displacements are not required to be positive, distinct, or in increasing order (however, negative displacements will precede the buffer)

Order of items need not coincide with their order in memory, and an item may appear more than once
**Type Map**

<table>
<thead>
<tr>
<th>Primitive datatype 0</th>
<th>Displacement of 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive datatype 1</td>
<td>Displacement of 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Primitive datatype n-1</td>
<td>Displacement of n-1</td>
</tr>
</tbody>
</table>
Extent

[Jargon]

- Extent: distance, in bytes, from beginning to end of type

- More specifically, the extent of a data type is defined as:
  … the span from the first byte to the last byte occupied by entries in this data type rounded up to satisfy alignment requirements

- Example:
  - Type={\texttt{(double},0),\texttt{(char},8)} i.e. offsets of 0 and 8 respectively.
  - Now assume that doubles are aligned strictly at addresses that are multiples of 8
  - extent = 16 (9 rounds to next multiple of 8, which is where the next double would land)
Map Type, Examples

- What is extent of type \{(char, 0), (double, 8)\}?  
  Ans: 16

- Is this a valid type: \{(double, 8), (char, 0)\}?  
  Ans: yes, since order does not matter
Example

- What is Type Map of `MPI_INT`, `MPI_DOUBLE`, etc.?
  - `{(int,0)}`
  - `{(double, 0)}`
  - Etc.
The sequence of primitive data types (i.e. displacements ignored) is the type signature of the data type.

Example: a type map of

\{(double,0),(int,8),(char,12)\}

...has a type signature of

\{double, int, char\}
Data Type Interrogators

- **datatype** - primitive or derived **datatype**
- **extent** - returns extent of **datatype** in bytes

```c
int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent);
```

- **datatype** - primitive or derived **datatype**
- **size** - returns size in bytes of the entries in the **type signature** of **datatype**
  - Gaps don’t contribute to size
  - This is the total size of the data in a message that would be created with this **datatype**
  - Entries that occur multiple times in the **datatype** are counted with their multiplicity

```c
int MPI_Type_size (MPI_Datatype datatype, int *size);
```
Committing Data Types

- Each derived data type constructor returns an *uncommitted* data type. Think of commit process as a compilation of data type description into efficient internal form.

```c
int MPI_Type_commit (MPI_Datatype *datatype);
```

- **Required** for any derived data type before it can be used in communication.

- Subsequently can use in any function call where an `MPI_Datatype` is specified.
MPI_Type_free

```c
int MPI_Type_free(MPI_Datatype *datatype);
```

- Call to `MPI_Type_free` sets the value of an MPI data type to `MPI_DATATYPE_NULL`.
- Data types that were derived from the defined data type are unaffected.
The inconsistent naming convention is unfortunate but carries no deeper meaning. It is a compatibility issue between old and new version of MPI.
MPI_Type_contiguous

```c
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- **IN** count (replication count)
- **IN** oldtype (base data type)
- **OUT** newtype (handle to new data type)

- Creates a new type which is simply a replication of old type into contiguous locations
```c
#include <stdio.h>
#include <mpi.h>

/* !!! Should be run with at least four processes !!! */
int main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
        int x;
        int y;
        int z;
    } point;
    MPI_Datatype ptype;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(3,MPI_INT,&ptype);
    MPI_Type_commit(&ptype);
    if( rank==3 ){
        point.x=15; point.y=23; point.z=6;
        MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
    }
    else if( rank==1 ) {
        MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
        printf("P:%d received coords are (%d,%d,%d) \n",rank,point.x,point.y,point.z);
    }
    MPI_Type_free(&ptype);
    MPI_Finalize();
    return 0;
}
```
Example: MPI_Type_contiguous

[Output]

[negrut@euler24 CodeBits]$ mpiexec -np 10 me759.exe
P:1 received coords are (15,23,6)

[negrut@euler24 CodeBits]$
Motivation: MPI_Type_vector

- Assume you have a 2D array of integers, and want to send the last column

```
int x[4][8];
```

Content of x:

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>1006</td>
<td>1007</td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10002</td>
<td>10003</td>
<td>10004</td>
<td>10005</td>
<td>10006</td>
<td>10007</td>
</tr>
</tbody>
</table>

- There should be a way to say that I want to transfer integers, 4 of them, and they are stored in array x 8 integers apart (the stride)
MPI_Type_vector: Example

- count = 2
- blocklength = 3
- stride = 5
MPI_Type_vector

- **MPI_Type_vector** is a constructor that allows replication of a data type into locations that consist of equally spaced blocks.

- Each block is obtained by concatenating the same number of copies of the old data type.

- Spacing between blocks is a multiple of the extent of the old data type.

- One way to look at it:
  - You want some entries but don’t care about other entries in an array.
  - There is a repeatability to this pattern of “wanted” and “not wanted” entries.
MPI_Type_vector (int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype);

- IN count (number of blocks)
- IN blocklength (number of elements per block)
- IN stride (spacing between start of each block, measured as # elements)
- IN oldtype (base datatype)
- OUT newtype (handle to new type)

Allows replication of old type into locations of equally spaced blocks. Each block consists of same number of copies of oldtype with a stride that is multiple of extent of old type.
#include <mpi.h>
#include <math.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);

    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,52,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)printf("P:%d my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }

    MPI_Type_free(&coltype);
    MPI_Finalize();
    return 0;
}
**Example: MPI_Type_vector**

**Output**

Content of x:

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>1006</td>
<td>1007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>10001</td>
<td>10002</td>
<td>10003</td>
<td>10004</td>
<td>10005</td>
<td>10006</td>
<td>10007</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[negrut@euler19 CodeBits]$ mpiexec -np 12 me759.exe
P:1 my x[0][2]=17.000000
P:1 my x[1][2]=107.000000
P:1 my x[2][2]=1007.000000
P:1 my x[3][2]=10007.000000
[negrut@euler19 CodeBits]$
Example: MPI_Type_vector

- Given: Local 2D array of interior size $m \times n$ with $n_g$ ghostcells at each edge
- You wish to send the interior (non ghostcell) portion of the array
- How would you describe the data type to do this in a single MPI call?

$$\text{MPI_Type_vector}(\text{int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype});$$

**Ans:**
$$\text{MPI_Type_vector} (m, n, n+2*ng, \text{MPI_DOUBLE}, &\text{interior});$$
$$\text{MPI_Type_commit} (&\text{interior});$$
$$\text{MPI_Send} (\text{startPoint}, 1, \text{interior}, \text{dest}, \text{tag}, \text{MPI_COMM_WORLD});$$
Type Map Example

- Start with `oldtype` for which
  Type Map = {((double, 0), (char, 8))}

- What is Type Map of `newtype` if defined as below?
  `MPI_Type_vector(2,3,4,oldtype,&newtype)`

**Ans:**

```
{{(double, 0), (char, 8)},
 ((double,16),(char,24) ),
 ((double,32),(char,40) ),
 ((double,64),(char,72) ),
 ((double,80),(char,88) ),
 ((double,96),(char,104))}
```
Exercise: MPI_Type_vector

- Express
  
  `MPI_Type_contiguous(count, old, &new);`
  
  ...as a call to `MPI_Type_vector`

  ```c
  MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
  ```

- Ans:
  
  ```c
  MPI_Type_vector (count, 1, 1, old, &new);
  MPI_Type_vector (1, count, count, old, &new);
  ```
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
In some MPI implementations there are more than 300 MPI functions

Not all of them part of the MPI standard though, some vendor specific

Recall the 20/80 rule: six calls is probably what you need to implement a decent MPI code...

- MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize
The PETSc Library
[The message: Use libraries if available]

- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)
Structure of PETSc

PETSc PDE Numerical Solution Utilities

- ODE Integrators
- Nonlinear Solvers, Unconstrained Minimization
- Linear Solvers, Preconditioners + Krylov Methods
- Object-Oriented Matrices, Vectors, Indices
- Grid Management
- Profiling Interface

Computation and Communication Kernels
MPI, MPI-IO, BLAS, LAPACK
# PETSc Numerical Components

## Nonlinear Solvers
- Newton-based Methods
- Line Search
- Trust Region
- Other

## Time Steppers
- Euler
- Backward Euler
- Pseudo Time Stepping
- Other

## Krylov Subspace Methods
- GMRES
- CG
- CGS
- Bi-CG-STAB
- TFQMR
- Richardson
- Chebychev
- Other

## Preconditioners
- Additive Schwartz
- Block Jacobi
- Jacobi
- ILU
- ICC
- LU (Sequential only)
- Others

## Matrices
- Compressed Sparse Row (AIJ)
- Blocked Compressed Sparse Row (BAIJ)
- Block Diagonal (BDIAG)
- Dense
- Matrix-free
- Other

## Distributed Arrays

## Vectors

## Index Sets
- Indices
- Block Indices
- Stride
- Other
Flow Control for PDE Solution

- Application Initialization
- Function Evaluation
- Jacobian Evaluation
- Post-Processing

User code

PETSc code

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PC

KSP

PETSc
CUDA, OpenMP, MPI: Putting Things in Perspective
Pros, CUDA

- Many remarkable success stories when the application targeted is data parallel and with high arithmetic intensity
  - One order of magnitude speed-ups are common

- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L got you six or seven years ago

- Ubiquitous
  - Present on more than 100 million computers today support CUDA

- Good productivity tools
Cons, CUDA

- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware.
- Not that much device memory available – 6 GB is the most you get today:
  - Getting around it requires moving data in and out of the device, which complicates the programming job.
- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance and complicating the implementation task.
- For true HPC, using CUDA in conjunction with MPI remains a challenge:
  - Ongoing projects aimed at addressing this, but still…
What Would Be Nice…

- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases.

- Integrate CPU & GPU so that concept of global device memory disappears.

- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming.
Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement
- Programming model is “serial-like”, thus conceptually simpler than message passing
- Compiler directives are generally simple and easy to use
- Legacy serial code does not need to be rewritten
Cons of OpenMP

- The model doesn’t scale up all that well
- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl’s Law prohibits substantial speedups
- Amdahl’s Law:
  \[ s = \text{Fraction of serial execution time that cannot be parallelized} \]
  \[ N = \text{Number of processors} \]

Execution speedup: \[ = \frac{1}{s + \frac{1-s}{N}} \]

- If you have big loops that dominate execution time, these are ideal targets for OpenMP
Pros of MPI

● Good vendor support for the standard
  ● It was great that the community converged upon a standard (something that can’t be said about GPU computing)

● Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores

● Can be deployed both for distributed as well as shared memory architectures

● Today it is synonym with High Performance Computing
  ● Provided a clear and relatively straightforward framework for reaching Petaflops grade computing
Cons of MPI

- The interconnect is Achilles' heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically worse though

- Like CUDA, works well only for applications where you don’t have to communicate all that much (high arithmetic intensity)
General Remarks on Parallel Computing

- Parallel Computing is and will be relevant at least for this decade

- Nonetheless, it continues to be challenging
  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it’s a skill that is eventually acquired
    - Parallel Programming more difficult than programming for Sequential Computing
  - Productivity tools (debuggers, profilers, build solutions) more challenging to master
  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run
Skills I hope You Picked Up in ME759

- I think of these as items that you can add to your resume:
  - Basic understanding of hardware for parallel computing
  - Basic understanding of parallel execution models: SIMD, MIMD, etc.
  - CUDA programming
  - OpenMP Programming
  - MPI Programming
  - [ Build management: Cmake ]
  - Debugging: gdb, cuda-gdb, memcheck, cuda-memcheck
  - Profiling: nvvp
ME759: Most Important Two Things

- Don’t move data around
  - Costly in terms of time and energy.

- Hone your “computational thinking” skills