GPU Computing with CUDA

Dan Negrut
Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

© Dan Negrut, 2012
UW-Madison

Milano
10-14 December 2012
Before We Get Started…

• Goal, GPU segment
  • Spend next three days getting familiar with GPU computing using CUDA
  • Understand whether you can put GPU computing and CUDA to good use

• Reaching this goal
  • Cover some basics (one day) and more advanced features (a second day)
  • Talk about library support & productivity tools (the third day)
    • thrust, cuda-gdb, nvvp
HPC: Where Are We Today?
[Info lifted from Top500 website: http://www.top500.org/]
Where Are We Today?

[Cntd.]

- **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)
How is the speed measured to put together the Top500?

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

HPLINPACK

A Portable Implementation of the High Performance Linpack Benchmark for Distributed Memory Computers

- Algorithm: recursive panel factorizations, multiple lookahead depths, bandwidth reducing swapping
- Easy to install, only needs MPI • BLAS or VSIPL
- Highly scalable and efficient from the smallest cluster to the largest supercomputers in the world

Find out more at http://icl.eecs.utk.edu/hpl/
Some Trends…

- Consequence of Moore’s law
  - Transition from a speed-based compute paradigm to a concurrency-based compute paradigm

- Amount of power for supercomputers is a showstopper
  - Example:
    - Exaflop/s rate: the goal is to 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:
    - Japan’s fastest computer (Kei): 12.7 MW for 10.5 Petaflop/s
    - China’s fastest supercomputer (Tianhe-1A): 4.0 MW for 2.6 Petaflop/s
    - US fastest supercomputer (Oak Ridge Jaguar’s): 8.3 MW for 17.6 Petaflop/s
    - Faster machine for less power: the advantage of GPU computing
GPU Computing is Power Efficient

- One Kepler card: 15.85 Gflop/W
- Japan’s Kei: 0.825 Gflops/W
- China’s Tianhe-1A: 0.636 Gflops/W
- USA’s Jaguar: 0.251 Gflops/W
- Best HPC cluster performance - IBM's NNSA/SC Blue Gene/Q Prototype 2: 2.097 GFlops/W
  - Currently the world's 109th-fastest supercomputer
- If we are to reach exascale by 2018: 5-50 Gflops/W
Why GPU Computing?

- It’s fast for a variety of jobs
  - Really good for data parallelism (which requires SIMD)
  - However, not impressive for task parallelism (which requires MIMD)

- It’s cheap to get one ($120 to $500)
  - High end GPUs for Scientific Computing are more like $1500

- GPUs are everywhere
  - There is incentive to produce software since there are many potential users of it…
  - More than 300 million NVIDIA CUDA enabled cards

- NOTE: GPU computing is not quite High Performance Computing (HPC)
  - However, it shares with HPC the aspect that they both draw on parallel programming
IBM BlueGene/L

- Entry model: 1024 dual core nodes
- 5.7 Tflop/s
- Linux OS
- Dedicated power management solution
- Dedicated IT support
- Price (2007): $1.4 million
Euler: Heterogeneous Cluster Used in This Tutorial
Euler: Heterogeneous Cluster Used in This Tutorial
Euler, Quick Overview

- More than 25,000 GPU scalar processors
  - Can manage about 75,000 GPU parallel threads at full capacity
- More than 1000 CPU cores
- Mellanox Infiniband Interconnect, 40Gb/sec
- About 2.7 TB of RAM
- More than 20 Tflops DP
- …
Euler, Quick Overview

- More than 25,000 GPU scalar processors
  - Can manage about 75,000 GPU parallel threads at full capacity
- More than 1000 CPU cores
- Mellanox Infiniband Interconnect, 40Gb/sec
- About 2.7 TB of RAM
- More than 20 Tflops DP
- …

The issues is not hardware availability. Rather, it is producing modeling and solution techniques that can leverage this hardware.
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

- 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?
Computational Science and Engineering

Old School  New School
Increasing clock frequency is primary method of performance improvement
Computational Science and Engineering

**Old School**

- Increasing clock frequency is primary method of performance improvement

**New School**

- Don’t count on frequency increases as main driver of your performance improvement
Computational Science and Engineering

Old School

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, parallel computing is odd and expensive

New School

- Don’t count on frequency increases as main driver of your performance improvement
Computational Science and Engineering

Old School

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, parallel computing is odd and expensive

New School

- Don’t count on frequency increases as main driver of your performance improvement
- Nobody builds one core processors anymore. Processors’ parallelism is the de-facto method for performance improvement.
Old School

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, parallel computing is odd and expensive
- Less than linear scaling for a multiprocessor is failure

New School

- Don’t count on frequency increases as main driver of your performance improvement
- Nobody builds one core processors anymore. Processors’ parallelism is the de-facto method for performance improvement.
Computational Science and Engineering

**Old School**

- Increasing clock frequency is primary method of performance improvement
- Don’t bother parallelizing an application, parallel computing is odd and expensive
- Less than linear scaling for a multiprocessor is failure

**New School**

- Don’t count on frequency increases as main driver of your performance improvement
- Nobody builds one core processors anymore. Processors’ parallelism is the de-facto method for performance improvement.
- Given the switch to parallel hardware, even sub-linear speedups are beneficial as long as you beat the sequential
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 \(O(N^{\frac{1}{2}} \log N)\)
    - 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 accordingly
  - 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
Layout of Typical Hardware Architecture

- CPU (the "host")
- GPU w/ local DRAM (the "device")
Bandwidth in a CPU-GPU System
# Key Parameters

## GPU, CPU

<table>
<thead>
<tr>
<th></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>$515 \times 10^9$ Double Precision</td>
<td>$70 \times 10^9$ Double Precision</td>
</tr>
</tbody>
</table>
GPU vs. CPU – Memory Bandwidth

[GB/sec]
CPU vs. GPU – Flop Rate (GFlops)

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Westmere 3 GHz
- Nehalem 3 GHz

GFlop/Sec

2003 2004 2005 2006 2007 2008 2009 2011
More Up-to-Date, DP Figures…

 GPUs DELIVER SUPERIOR PERFORMANCE IN PARALLEL COMPUTATION.

 GPUs deliver superior performance in parallel computation.

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

1. 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)

2. The fast-growing video game industry exerts strong economic pressure that forces constant innovation
CUDA: Making the GPU Tick…

- “Compute Unified Device Architecture” – freely distributed by NVIDIA

- 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 and graphics-free API
  - Explicit GPU memory management
CUDA Programming Model:
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
Next Two Slides Are Important
GPU: Underlying Hardware

- NVIDIA nomenclature used below reminiscent of GPU’s mission

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - …has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)

\[
\text{SPA} \xrightarrow{\text{has}} \frac{10}{10} \xrightarrow{} \text{TPC} \xrightarrow{\text{each has}} \frac{3}{3} \xrightarrow{} \text{SM} \xrightarrow{\text{each has}} \frac{8}{8} \xrightarrow{} \text{SP}
\]
NVIDIA TESLA C1060

- 240 Scalar Processors
- 4 GB device memory
- Memory Bandwidth: 102 GB/s
- Clock Rate: 1.3GHz
- Approx. $1,250

- The most important component of a GPU is the SM (Stream Multiprocessor)
- It is the quantum of scalability
“Compute Capability of a Device” refers to **hardware**
- Defined by a major revision number and a minor revision number

**Example:**
- Newton’s Tesla C1060 is compute capability 1.3
- Tesla C2050 is compute capability 2.0
- The major revision number is up to 3 (Kepler architecture)
- 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 5.0

The best setup
- You run the most recent CUDA (version 5.0) software release
- You use the most recent architecture (compute capability 3.0)
## 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>GTS 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,</td>
<td>2.1</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>GT 420M</td>
<td></td>
<td></td>
<td></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,</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 275</td>
<td></td>
<td></td>
<td></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,</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>9800 GTX, 9800 GTX+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX+, 8800 GTS 512,</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>GTX 285M, GTX 280M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8800 Ultra, 8800 GTX</td>
<td></td>
<td></td>
<td></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
  - 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 = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>>(myimage);
```
Example: Hello World!
Example: Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```
Example: Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
Example: Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
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
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

Output, on Euler:
```
$ nvcc hello_world.cu
$ a.out
Hello World!
$ 
```
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

Output, on Euler:
```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```
Compiling CUDA

- Source files with CUDA language extensions must be compiled with `nvcc`
  - You spot such a file by its .cu suffix

- Example:
  ```
  >>> nvcc -arch=sm_20 foo.cu
  ```

- Actually, `nvcc` is 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 ISA)
  - 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

CUDA C/C++ keyword __global__ indicates a function that:
- Runs on the device
- Is called from host code

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

```c
__global__ void mykernel(void) {
}
```
Hello World! with Device Code

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - NOTE: we’ll return to the parameters (1,1) 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;
}
```

- Actually, `mykernel()` does not do anything yet...

Output, on Euler:
```
$ nvcc hello.cu
$ a.out
Hello World!
$
```
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)

| __device__ | float myDeviceFunc() | device | device |
| __global__ | void myKernelFunc()  | device | host   |
| __host__   | float myHostFunc()   | host   | host   |

- __global__ defines a kernel function, launched by host, executed on the device
  - Must return **void**
- For a full list, see CUDA Reference Manual
The Concept of Execution Configuration

- A kernel function must be called with an **execution configuration**:

  ```
  __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...);
  ```

- 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”

```
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 the decide what data to work on**
  - Block Index: a pair of uint
  - Thread Index: a triplet of three uint

- Why this 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on subdomains
    - ...

![Diagram](https://via.placeholder.com/150)

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 the block and thread indices

- Each thread when executing a *device* function 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 understanding how SIMD is supported in CUDA &
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 on Tesla C1060 cannot have more than 512 threads $\Rightarrow D_x \times D_y \times D_z \leq 512$.

  - Note: On Fermi and Kepler architectures this is 1024.

- Each thread in a block can be identified by a unique index $(x, y, z)$, and

  $0 \leq x \leq D_x \quad 0 \leq y \leq D_y \quad 0 \leq z \leq D_z$

- 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 $D_y = D_z = 0$. Handling matrices typically goes well with $D_z = 0$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.
A Recurring Theme in CUDA Programming
[and in SIMD in general]

- Imagine you are one of many threads, and you have your thread index and block index

  - You need to figure out what is the work you need to do
    - 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
#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;
}
```cpp
#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;
}
```
Review - Execution Configuration: Grids and Blocks

• A kernel is executed as a grid of blocks of threads
  • All threads in a kernel can access several device data memory spaces

• A block [of threads] is a batch 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?

[Graphics of grid and block structure]
Example: Adding Two Matrices

- You have two matrices A and B of dimension N£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 * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
Something to think about…

• Given that the device operates with groups of threads of consecutive ID, and given the scheme a few slides ago to compute a thread ID based on the thread & block index, 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…

  \[
  C[j][i] = A[j][i] + B[j][i]
  \]
Example: Array Indexing

- Purpose of Example: see a scenario of how multiple blocks are used to index entries in an array

- Recall that there is a limit on the number of threads you can have in a block

- In the vast majority of applications you need to use many blocks, each containing the same number of threads
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries

```
blockIdx.x = 0  blockIdx.x = 1  blockIdx.x = 2  blockIdx.x = 3
```
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries

```
threadIdx.x  threadIdx.x  threadIdx.x  threadIdx.x
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```

```
blockIdx.x = 0  blockIdx.x = 1  blockIdx.x = 2  blockIdx.x = 3
```

[NVIDIA]→
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries

- With $M$ threads/block a unique index for each thread is given by:
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries

With $M$ threads/block a unique index for each thread is given by:
Example: Array Indexing
[Important to Grasp]

- 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/block and the array has 32 entries

```c
int index = threadIdx.x + blockIdx.x * M;
```

- With $M$ threads/block a unique index for each thread is given by:
Example: Array Indexing

What will be the array entry that thread of index 5 in block of index 2 will work on?
Example: Array Indexing

- What will be the array entry that thread of index 5 in block of index 2 will work on?

M = 8

threadIdx.x = 5

blockIdx.x = 2
What will be the array entry that thread of index 5 in block of index 2 will work on?

```c
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Example: Array Indexing

- What will be the array entry that thread of index 5 in block of index 2 will work on?

```c
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Example: 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.
#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
", currentDevice, deviceProp.name);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms
";

    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 the old GPGPU way of programming the hardware
  • CUDA API is exposed to you (the user) through a collection of header files
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 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

```c
void *cudaMalloc(size_t size)
{ /* Allocate memory on device */ }

void cudaFree(void *ptr)
{ /* Free memory on device */ }
```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
Example Use: A Matrix Data Type

- NOT part of CUDA API, but discussed here since used in several code examples

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Example Use: A Matrix Data Type

- NOT part of CUDA API, but discussed here since used in several code examples

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

- New type abstracts the following concept:
  - 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
Example
CUDA Device Memory Allocation (cont.)

● Code example:
  ● Allocate a 32 * 32 single precision float array
  ● Attach the allocated storage to \texttt{Md.elements}
  ● “d” in “Md” is often used to indicate a device data structure

\begin{verbatim}
BLOCK_SIZE = 32;
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);
\end{verbatim}
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 32 * 32 single precision float array
  - \( M \) is in host memory and \( M_d \) is in device memory
  - \texttt{cudaMemcpyHostToDevice} and \texttt{cudaMemcpyDeviceToHost} are symbolic constants

\[
\texttt{cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);}\\
\texttt{cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);}\]
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)
  - Concentrate on
    - Thread ID usage
    - Memory data transfer API between host and device
  - Assume all matrices are square, of dimension WIDTH=32
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $\text{WIDTH} \times \text{WIDTH}$
- **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 $N$
    - $2\times\text{WIDTH}$ 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;
        }
    }
}
Step 1: Matrix Multiplication, Host-side.
Main Program Code

```c
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;
}
```
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);
}

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;
}
// 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);
    cudaMalloc((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);
}
CUDA runtime API: exposes a set of extensions to the C language

It consists of:

- **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
Overview of Large Multiprocessor Hardware Configurations

- 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
      - Uniform cluster:
        - IBM BlueGene
      - Constellation cluster of DSMs or SMPs
        - SGI Altix, ASC Purple
      - Noncache coherent:
        - Cray T3E, X1
      - Cache coherent:
        - ccNUMA:
          - SGI Origin/Altix

© 2007 Elsevier, Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Overview of Large Multiprocessor Hardware Configurations

- 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

Euler

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Parallel Computing on a GPU

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

- Tesla C2050 delivers 0.515 Tflops in double precision

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - “Extended C”

- Tesla C2050

- Tesla C1060