Advanced Computing for Engineering Applications

Dan Negrut

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

Darmstadt University
August 31, 2016
Before we get started: Recap of what was cover yesterday

- Discussion related to parallel computing issues
  - Three walls to sequential computing
  - Flynn’s taxonomy
  - Amdahl's law
  - Scaling attribute

- Started discussion about GPU computing
  - The big picture
  - Execution configuration
  - Timing a CUDA kernel
  - Scheduling issues
  - Thread divergence issues
Before we get started

● Today
  ● The NVIDIA GPU memory ecosystem
  ● Atomic operations
  ● Optimization issues
    ● Occupancy issues
    ● Rules of thumb

● GPU computing with thrust

● Today’s challenge problem: sum up all the integers in a large array of up to 10 million elements
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
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** between 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 here.
The Concept of Local Memory

- Local memory does not exist physically
  - “Local” in scope but not in location
- Data that is stored in “local memory” is actually placed in cache or the global memory at run time or by the compiler.
  - 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 got local scope; i.e., it’s specific to one thread
  - Long access times for local memory (on 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</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</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: *fast* if cached, otherwise very slow
- Global Memory – DRAM: *slow* (unless if cached)
- 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 unless cached)

- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: intra-block inter-thread communication

- **Global Memory**: per-application basis
  - Available for use by all threads
  - Used for: global access, all threads
  - Also used for inter-grid communication

---

- Grid 0
- Grid 1
- Global Memory
- Sequential Grids in Time
Coming Up Next

- Talk about these three memories
  - Registers
  - Shared Memory
  - Global Memory
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 and Maxwell and Pascal

- 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 and Maxwell: a thread can have assigned by the compiler up to 255 registers

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example [Revisited]

- Purpose
  - See an example where the use of multiple blocks of threads plays a central role
  - Understand, through an example, the use/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
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
Answering the Question “Shared Memory Relevant?”

- Test whether shared memory is relevant or not
  - Imagine you are a thread and execute the kernel
  - If data that you use turns out that can be used by any other thread in your block then you should consider using shared memory

- Note: you can use shared memory as scratch pad memory
  - Don’t let it go wasted… use it just like you’d use registers
Multiply Using Several Blocks

- One **block** computes one square sub-matrix \( C_{sub} \) of size \( \text{Block}_\text{Size} \)

- One **thread** computes one entry of \( C_{sub} \)

- **Assumption:** \( A \) and \( B \) are *square matrices* and their dimensions of 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 \( \text{Block}_\text{Size}=16\times16 \)

NOTE: 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
A Block of 16 X 16 Threads
// 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);
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // 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);
}

(continues below…)

(continues with next block…)

Code on the host side
First entry of the tile (number of tiles along the width of B)

A

bBegin

bStep

A

aBegin

aStep

C

(by (number of tiles down the height of A))

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;

    // 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];

    // 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;
}

// 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) {
    // 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:
  - Cache 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
Sh 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 pool 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
Memory Issues Not Addressed Yet...

- Not all *global* memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question
  - Discussed next

- Not all *shared* memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions
  - Not discussed here
Global Memory Access Issues
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 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.
Fermi Memory Layout
[credits: NVIDIA]
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 integers can be brought over in fell swoop

- 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
More Memory Facts
[Fermi GPUs]

- 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
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

- Good to know: any address of memory allocated with `cudaMalloc` is a multiple of 256
  - That is, the address 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 significant slow down of the code

- 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?
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);
}
```
Test Your Understanding

- 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...

\[
C[j][i] = A[j][i] + B[j][i]
\]
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
Atomic Operations
[provides opportunity to understand why parallel computing is tricky]
Coordinating Memory Operations

- Accesses to shared locations (global memory & shared memory) need to be correctly coordinated (orchestrated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination mechanisms such as locks to choreograph accesses to shared data.

- CUDA has a scalable coordination mechanism called “atomic memory operation.”

Next slides: several examples that show “race conditions”
Recall, First CUDA Example

```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;
}
```
Test Your Understanding

- In our code, we invoked the kernel like this:
  \texttt{simpleKernel<<<1,4>>>(devArray)}

- What would happen if we invoke the kernel like this:
  \texttt{simpleKernel<<<2,4>>>(devArray)}
Another Race Condition

- A contrived (artificial) example...

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
  int i = threadIdx.x;
  if (i < 2)
    *x = *y;
  else
    *x += 2*i;
}

// main.cpp
update_race<<<1,4>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Relevant Issue: Thread Divergence in “if-then-else”

- Handling of an if-then-else construct in CUDA
  - First a subset of threads of the warp execute the “then” branch
  - Next, the rest of the threads in the warp execute the “else” branch
    - Note: done if there is an “else” branch
Question: what happens if in the previous example we change the “if(i<2)” to “if(i>=2) ...” and swap the then and the else parts?
- Is the outcome any different?
- Any difference in behavior compared to sequential computing?

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x;
    if (i < 2)
        *x = *y;
    else
        *x += 2*i;
}
```

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x;
    if (i >= 2)
        *x += 2*i;
    else
        *x = *y;
}
```
Another Example

```c
#include <cuda.h>
#include "stdio.h"

global__ void testKernel(int *x, int *y) {
    int i = threadIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

int main() {
    int* dArr;
    int hArr[2] = {23, -5};
    cudaMalloc(&dArr, 2 * sizeof(int));
    cudaMemcpy(dArr, hArr, 2 * sizeof(int), cudaMemcpyHostToDevice);
    testKernel <<<1, 2 >>>(dArr, dArr + 1);
    cudaMemcpy(hArr, dArr, 2 * sizeof(int), cudaMemcpyDeviceToHost);
    printf("x = %d\n", hArr[0]);
    printf("y = %d\n", hArr[1]);
    return 0;
}
```
C:\Users\negrut\Bin\US13Projects\Cuda Sandbox\Darmstadt\Debug>testRace
x = 1
y = 1
C:\Users\negrut\Bin\US13Projects\Cuda Sandbox\Darmstadt\Debug>
Example: Inter-Block Issue

- Would this fly?

```c
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x;
    if (i == 0) *x = blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update<<<2,5>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
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
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, i);  // j is now old value of x;
}
```

Other 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 `atomicOP(a,b)` is typically

```
t1 = *a;    // read
```
```
t2 = (*a) 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.

- While order is not clear, none of the atomic updates will be lost.

- Several 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
//
// picturePixels – pointer to picture pixels, each w/ its own color
// bucket–pointer to histogram bucket of size equal to # of colors
//
__global__ void histogram(int n, int* picturePixels, int* bucket) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
        int c = picturePixels[i];
        atomicAdd(&bucket[c], 1);
    }
}
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…
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 += 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 because of possible race conditions

- Use for infrequent, sparse, and/or unpredictable global communication

- Use shared memory and/or customized data structures & algorithms to avoid synchronization whenever reasonable

- Recent compute capabilities (3 and above) implement very fast atomics
As far as CUDA is concerned, there is a qualitative difference between a `__syncthreads()` function and an atomic operation.

- `__syncthreads()` has the connotation of barrier; i.e., of synchronization.
  - `__syncthreads()` establishes a point in the execution of the kernel that every thread in the **block** needs to reach before any block thread can move beyond that point.

- The “atomic operation” concept instead tied to the idea of coordination in relation to operations that involve memory transactions.
  - Threads in a grid of blocks coordinate their execution so that a certain memory operation invoked in a kernel is conducted in an atomic fashion.
CUDA GPU Programming
~ Resource Management Considerations ~
What Do I Mean By “Resource Management”?

- The GPU is a resourceful device

- What do you have to do to make sure you capitalize on these resources?
  - In other words, how can you ensure that the SM hardware is used at capacity?
  - “used at capacity”: the SM executes the maximum number of warps that it possibly can

- The three factors that come into play are
  - How many threads you decide to use in each block
  - What register requirements end up associated with a thread
  - How much shared memory you assign to one block of threads
Some Hard Constraints  [1 of 2]

- Max number of warps that one SM can service simultaneously:
  - 32 on Tesla C1060, 48 on Fermi, 64 on Kepler

- Max number of blocks that one SM can process simultaneously:
  - 8 (Fermi), 16 (Kepler), 32 (Maxwell)
Some Hard Constraints [2 of 2]

- The number of 32-bit registers available on each SM is limited:
  - 16,384 registers (on Tesla C1060)
  - Roughly 48,000 on Fermi
  - Roughly 64,000 on Kepler and Maxwell

- The amount of shared memory available to each SM is limited
  - 16 KB on Tesla 1060
  - 64 KB on Fermi (16/48 or 48/16 configurable between L1$ and shared memory)
  - 64 KB on Kepler (16/48 or 48/16 or 32/32 configurable)
  - 64 KB on Maxwell, but not split w/ L1$
The Concept of Occupancy
[Discussion Focused on Tesla]

- Ideally, you want to have 32 warps serviced at the same time by one SM
  - That is, you’d like to have the SM end up managing the max number of warps that it is designed to handle
  - This keeps the SM busy and hides latencies associated with memory access

- Examples, for Fermi:
  - Two blocks with 512 threads running together on one SM: 100% occupancy
  - Four blocks of 256 threads each running on one SM: 100% occupancy
  - 16 blocks with 64 threads each – not good, can’t have more than 8 blocks running on a SM
    - Effectively this scenario gives you at most 50% occupancy
The Concept of Occupancy

- What prevents you from getting high occupancy?
  - Amount of shared mem demanded by each block
    - Total amount of shared memory in one SM is limited: up to 64 Kb on Maxwell
  - Number of registers used by each thread
    - Size of the register file in one SM: 64K four byte registers on Kepler and Maxwell

- If you ask for too much shared memory or use too many registers you can have 32 warps active on the SM; i.e., can’t have 100% occupancy
Examples, Occupancy of HW

- **Example 1, Fermi**: If each of your block demands 80 KB of shared memory, the kernel will fail to launch
  - Not enough memory on the SM to run even a block

- **Example 2, Fermi**: If your blocks each uses 15 KB of shared mem, you can have up to three blocks running on one SM (there will be some shared mem that will go unused)

- **Example 3, Fermi**: Like Example 2 above, and you have 512 threads per block, each thread uses 20 registers. Will one SM be able to handle 2 blocks?
  - Total number of registers: $512 \times 2 \times 20 = 20,480$ out of the 32,000 are used) ok
  - Amount of shared memory: $2 \times 15K = 30$ KB, well below 48 KB
  - Number of warps: 2 blocks $\times$ 512 threads $= 1024$ threads $= 32$ warps $< \text{max of 48}$, ok
  - Question: Will the SM be able to handle 3 blocks?
Resource Utilization

- There is an “occupancy calculator” that can tell you what percentage of the HW gets utilized by your kernel

- Assumes the form of an Excel spreadsheet

- Requires the following input
  - Threads per block
  - Registers per thread
  - Shared memory per block

- google “occupancy calculator cuda” to access it
Granularity Considerations

[NOTE: Specific to Fermi]

- For Matrix Multiplication example (with shared memory), 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 32 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
CUDA Optimization: Wrap Up…
Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code. Expose fine grain parallelism

2. 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

Writing CUDA Software: High-Priority Recommendations

4. Strive to have aligned and coalesced global memory accesses. Design your implementation such that global memory accesses are coalesced for that part of the red-hot parts of the code.

5. Minimize the use of global memory. Prefer shared memory access where possible (consider tiling as a design solution).

Writing CUDA Software: Medium-Priority Recommendations

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

Writing CUDA Software: Medium-Priority Recommendations

4. Use the fast math library whenever speed is very important and you can live with a tiny loss of accuracy

5. Avoid thread divergence

CUDA GPU Code Development
Code Development Support

- How do I compile?
- How do I link?
- How do I debug?
- How do I profile?
Compiling CUDA Code

[with nvcc driver]
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;
    }
}
```
Debugging Tools

- **cuda-gdb**
  - Used to step through the code
  - Can use to select which thread you want to run as
  - Available under Linux
  - There is a Visual Studio debugger as well

- **cuda-memcheck**
  - Use to check the sanity of your memory use
  - Use even if the results seem to be correct
Approach 1: command-line profiler `nvprof`
- Collect timeline of CPU and GPU activities
- Headless profile collection
  - Use `nvprof` on headless node to collect data
  - Visualize timeline with Visual Profiler

Approach 2: 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)
**nvprof Usage**

$ nvprof [nvprof_args] <app> [app_args]

- Help on usage and arguments:
  $ nvprof --help

- Save profile to file that can be later import for post-processing:
  $ nvprof -o profile.out <app> [app_args]
  - Option 1: Import into NVIDIA Visual Profiler (nvvp):
    - File menu -> Import nvprof Profile
  - Option 2: Import into nvprof to generate reports:
    $ nvprof -i profile.out
    $ nvprof -I profile.out --print-gpu-trace
nvprof – GPU Summary

$ nvprof vector_addition

- Generate CSV output
  $ nvprof --csv vector_addition
nvprof – GPU Trace

$ nvprof --print-gpu-trace vector_addition

- CPU/GPU trace
  $ nvprof --print-gpu-trace --print-api-trace vector_addition
nvvp: NVIDIA Visual Profiler

- Available on Euler
- Provides a nice GUI and ample information regarding your run
- Many bells & whistles
Further Information

More resources:
- CUDA tutorials video/slides at GTC
- CUDA webinars covering many introductory to advanced topics

Other related topic:
- Performance Optimization Using the NVIDIA Visual Profiler
GPU COMPUTING WITHOUT KERNELS
Three Ways to Accelerate on GPU

Application

Libraries
Directives
Programming Languages

Easiest Approach  Maximum Performance

Direction of increased performance (and effort)

NVIDIA [C. Woolley]→
Next: Focus on Libraries

- Example library: **thrust**

- Acknowledgement:
  - 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

- Adopt a generic programming angle
  - Leverage a template-based approach

- Should run fast
  - Efficient mapping to hardware
What is **thrust**?

- A template library for CUDA
  - Mimics the C++ STL

- Relies heavily on use of containers
  - On host and device

- Provides a series of ready to use generic 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. is all gone.
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // generate about 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 (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;
}
Namespaces, containers, iterators
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());
```
Containers

- Make common operations concise and readable

```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;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;

// vector memory automatically released w/ free() or cudaFree()
```
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());
```
Iterators

- Sequences are defined by pairs of iterators
- For vector containers, iterators act like pointers
- They can be used like pointers (e.g. incremented)
- But encapsulate more information than pointers
  - Permit tracking memory space (host/device)
  - Guide algorithm dispatch (static dispatching)
- Can be converted to raw pointers

```c++
// allocate device vector
thrust::device_vector<int> d_vec(4);
d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end() // returns iterator one past the last element of d_vec
```
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 / 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);
```
Namespaces, containers, iterators

- **Namespaces**
  - Avoids collisions

- **Containers**
  - Manage host & device memory
  - Automatic allocation and deallocation
  - Simplify data transfers

- **Iterators**
  - Behave like pointers
  - Keep track of memory spaces
  - Convertible to raw pointers
Algorithms
Algorithms

- Element-wise 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` ...
Thrust Example: Sort

```c++
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void) {
    // 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>
Input-Sensitive Optimizations

Sorting Rate (Mkey/s) vs. Key Bits

NVIDIA [N. Bell]→
Example: Vector Addition

\[
\text{for (int } i = 0; i < N; i++) \\
Z[i] = X[i] + Y[i];
\]
#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

[serban@lagrange:~/CODES/GPU]$ nvcc -o check_props check_props.cu
[serban@lagrange:~/CODES/GPU]$ ./check_props
Device Number: 0
   Device name: Tesla K40c
   Memory Clock Rate (KHz): 3004000
   Memory Bus Width (bits): 384
   Peak Memory Bandwidth (GB/s): 288.384000

[serban@lagrange:~/CODES/GPU]$ nvcc -O3 -o vec_add vector_addition.cu
[serban@lagrange:~/CODES/GPU]$ ./vec_add
Adding vectors of length = 10000000
Result is correct.
Elapsed time: 0.723872

- Note: file extension should be .cu
#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 = X[0];

    float result = thrust::reduce(X.begin(), X.end(),
                                   init,
                                   thrust::maximum<float>());

    std::cout << "maximum is " << result << "\n";

    return 0;
}
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>());
```
Custom Types & Operators

```
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);
```
Custom Types & Operators

```cpp
// 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

```cpp
// 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);
```
Example: SAXPY

```
for (int i = 0; i < N; i++)
    Z[i] = a * X[i] + Y[i];
```
struct saxpy
{
    float m_a;

    saxpy(float a) : m_a(a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return m_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;
}
<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>

NVIDIA [N. Bell]→
General transformations
Zipping & fusing
Fancy iterators

- Inspired by and derived from those in the Boost Iterator Library
- Perform a variety of **valuable purposes**

- `constant_iterator` – returns same value whenever accessed
- `counting_iterator` – returns a sequence of increasing values
- `transform_iterator` – allows kernel fusion even when no algorithm `transform_xxx` is available
- `permutation_iterator` – allows fusing gather and scatter operations with Thrust algorithms
- `zip_iterator` - takes multiple inputs and returns a sequence of tuples
  - Allows implementation of arbitrary transformations
  - Performance advantage: SoA approach for coalesced memory access
General Transformations

- **Unary Transformation**
  
  ```
  for (int i = 0; i < N; i++)
  X[i] = f(A[i]);
  ```

- **Binary Transformation**
  
  ```
  for (int i = 0; i < N; i++)
  X[i] = f(A[i], B[i]);
  ```

- **Ternary Transformation**
  
  ```
  for (int i = 0; i < N; i++)
  X[i] = f(A[i], B[i], C[i]);
  ```

- **General Transformation**
  
  ```
  for (int i = 0; i < N; i++)
  X[i] = f(A[i], B[i], C[i], ...);
  ```

- Like C++ STL, **thrust** provides built-in support for unary and binary transformations.
- Transformations involving 3 or more input ranges must use a different approach.
The Zipping Operation

Multiple Distinct Sequences

Unique Sequence of Tuples
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); // U <- 2X + 3Y + 4Z

    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;
}
```

Functor Definition

These are the important parts: three different entities are zipped together in one big one.
Example: thrust::transform_reduce

```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;
}
Performance Considerations
[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

- Tesla C2050
  - 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

NVIDIA [N. Bell]
## Arithmetic Intensity

### Kernel FLOP/Byte*

<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>Reduce</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

### Hardware**

<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

```c
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: reorganizes **data** for **thrust** processing
  - Fusing: reorganizes **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;
}
Fusing Transformations

Original Implementation

```
GPU 12 Bytes 4 Bytes 12 Bytes 4 Bytes
      |        |        |
      v        v        v
DRAM```

Optimized Implementation

```
GPU 12 Bytes
      |        |
      v        v
DRAM 8 Bytes```

- Since the operation is completely memory bound the expected speedup is $\approx 1.6x = 32/20$
Fusing Transformations

for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];

for (int i = 0; i < N; i++)
    sum += F(X[i]);

Loop Fusion
#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

Original Implementation

- GPU
- 4 Bytes
- GPU
- 4 Bytes
- 4 Bytes
- DRAM

Optimized Implementation

- GPU
- 4 Bytes
- DRAM

Try to answer this: how many times will we be able to run faster if we fuse?
Thrust wrap-up
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 Wrap-Up

- Significant boost in productivity at the price of small performance penalty
  - No need to be aware of execution configuration, shared memory, etc.

- Key concepts
  - Functor
  - Zipping data
  - Fusing operations

- Why not always use thrust?
  - There is no “perform finite element analysis” support in thrust.
  - Thrust provides support for primitives – up to us to use them as needed
thrust on GitHub

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)

http://thrust.github.io/

What is Thrust?

Thrust is a parallel algorithms library which resembles the C++ Standard Template Library (STL). Thrust’s high-level interface greatly enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs. Interoperability with established technologies (such as CUDA, TBB, and OpenMP) facilitates integration with existing software. Develop high-performance applications rapidly with Thrust!

Recent News

- Thrust v1.8.0 release (18 Mar 2015)
- Thrust v1.7.0 release (12 Jan 2015)
- Thrust v1.7.0 release (02 Jul 2013)
- Thrust Content from GTC 2012 (12 May 2012)
- Thrust v1.6.0 release (07 Mar 2012)
- Thrust v1.5.1 release (30 Jan 2012)
- Thrust v1.5.0 release (28 Nov 2011)
- Thrust v1.3.0 release (05 Oct 2010)

Examples

Thrust is best explained through examples. The following source code generates random numbers serially and then transfers them to a parallel device where they are sorted.

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
thrust in “GPU Computing Gems”

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 MOTIVATION

With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C/C++ (henceforth CUDA C) is an important feature; it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware. For this class of computational tasks CUDA C is an excellent solution. Thrust (1) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. By capturing the programmer’s intent at a high level, Thrust has the description to make informed decisions on the best mapping and parallel implementation.

PDF available at http://goo.gl/adj9S
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

\[
\begin{bmatrix}
y_1 \\ y_2 \\ y_3 \\ y_4
\end{bmatrix} = \alpha \begin{bmatrix}
2 & -1 \\
4 & -1 \\
5 & 9 & 1 \\
-1 & 8 & 3
\end{bmatrix} \begin{bmatrix}
-1 \\
2 \\
1 \\
3
\end{bmatrix} + \beta \begin{bmatrix}
2 \\
0 \\
-1 \\
2
\end{bmatrix}
\]
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

NVIDIA [C. Woolley]→
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

Mathematical expressions:

\[ F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi(x\frac{n}{N})} \]

\[ f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x\frac{n}{N})} \]

NVIDIA [C. Woolley]→
Speedups Compared to Multi-Threaded CPU Execution

![cuFFT Speedup Chart]

- 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
Advanced Memory Issues
Issues Discussed

- Zero-copy memory in CUDA
- Unified Virtual Addressing
- Managed Memory

Based on Dr. Dobbs article of Sept 30, 2014
Summary / Objective

- Premise: Managing and optimizing host-device data transfers has been challenging

- Key point: Unified Memory (UM) support in CUDA 6 simplifies the programmer’s job

- This segment’s two goals:
  - Briefly review history of CUDA host/device memory management
  - Explain how UM makes host/device memory management easier and more efficient
cudaMemcpy

- A staple of CUDA, available in release 1.0

- Setup was simple: one CPU thread dealt with one GPU
  - The drill:
    - Data transferred from host memory into device memory with cudaMemcpy
    - Data was processed on the device by invoking a kernel
    - Results transferred from device memory into host memory with cudaMemcpy

- Memory allocated on the host with malloc
- Memory allocated on the device using the CUDA runtime function cudaMemcpy
- The bottleneck: data movement over the PCI-E link
The PCI-E Pipe, Putting Things in Perspective

- PCI-E
  - V1: 3 GB/s (per direction)
  - V2: 6 GB/s
  - V3 (today): 12 GB/s

- Bandwidths above pretty small, see for instance
  - Host memory bus (25 – 51.2 GB/s per socket)
  - GMEM bandwidth 100 – 200 GB/s
cudaHostAlloc: A friend, with its pluses and minuses

- Host/Device data transfer speeds could be improved if host memory was not pageable
  - Rather than allocating with malloc, host memory was allocated using CUDA’s cudaHostAlloc()
  - No magic on the hardware side, data still moves back-and-forth through same PCI-E connection

- cudaHostAlloc cons
  - cudaHostAlloc-ing large amounts of memory can negatively impact overall system performance
    - Why? It reduces the amount of system memory available for paging
    - How much is too much? Not clear, dependent on the system and the applications running on the machine
  - cudaHostAlloc is slow - ballpark 5 GB/s
    - Allocating 5 GB of memory is timewise comparable to moving that much memory over the PCI-E bus
Key Benefits, cudaHostAllocating Memory

- Three benefits to replacing host malloc call with CUDA cudaHostAlloc call
  1. Enables faster device/host back-and-forth transfers
  2. Enables the use of asynchronous memory transfer and kernel execution
     - Draws on the concept of CUDA stream, a topic not covered here
  3. Enables the mapping of the pinned memory into the memory space of the device
     - Device now capable to access data on host while executing a kernel or other device function
- Focus next is on 3 above
Zero-Copy (Z-C) GPU-CPU Interaction

- Last argument ("flag") controls the magic:
  ```
  cudaError_t cudaHostAlloc ( void** pHost, size_t size, unsigned int flag)
  ```

- "flag" values: cudaHostAllocPortable, cudaHostAllocWriteCombined, etc.

- The "flag" of most interest is "cudaHostAllocMapped"
  - Maps the memory allocated on the host in the memory space of the device for direct access

- What's gained:
  - The ability to access a piece of data from pinned and mapped host memory by a thread running on the GPU without a CUDA runtime copy call to explicitly move data onto the GPU
  - This is called zero-copy GPU-CPU interaction, from where the name "zero-copy memory"
  - Note that data is still moved through the PCI-E pipe, but it’s done in a transparent fashion
Z-C, Further Comments

- More on the “flag” argument, which can take four values:
  - Use `cudaHostAllocDefault` argument for getting plain vanilla pinned host memory (call becomes identical in this case to `cudaMallocHost` call)
  - Use `cudaHostAllocMapped` to pick up the Z-C functionality
  - See documentation for `cudaHostAllocWriteCombined` the `cudaHostAllocPortable`
    - These two flags provide additional tweaks, irrelevant here

- The focus **should not be** on `cudaHostAlloc()` and the “flag”
  - This function call is only a means to an end

- Focus **should be** on the fact that a device thread can directly access host memory
From Z-C to UVA: CUDA 2.2 to CUDA 4.0

- Z-C enabled access of data on the host from the device required one additional runtime call to `cudaHostGetDevicePointer()`
  - `cudaHostGetDevicePointer()`: given a pointer to pinned host memory produces a new pointer that can be invoked within the kernel to access data stored on the host

- The need for the `cudaHostGetDevicePointer()` call eliminated in CUDA 4.0 with the introduction of the Unified Virtual Addressing (UVA) mechanism
Unified Virtual Addressing: CUDA 4.0

- CUDA runtime can identify where the data is stored based on the value of the pointer
  - Possible since one address space was used for all CPU and GPU memory

- In a unified virtual address space setup, the runtime manipulates the pointer and allocation mappings used in device code (through cudaMalloc) as well as pointers and allocation mappings used in host code (through cudaHostAlloc()) inside a single unified space
UVA - Consequences

- There is no need to deal with cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, and cudaMemcpyDeviceToDevice scenarios
  - Simply use the generic cudaMemcpyDefault flag

- Technicalities regarding the need to call cudaMemcpyDeviceProperties() for all participating devices (to check cudaDeviceProp::unifiedAddressing flag) to figure out whether they’re game for UVA are skipped

- What this buys us: ability to do, for instance, inter-device copy that does not rely on the host for staging data movement:
  - cudaMemcpy(gpuDst_memPtr, gpuSrc_memPtr, byteSize, cudaMemcpyDefault)
UVA – Showcasing Its Versatility…

- Set of commands below can be issued by one host thread to multiple devices
  - No need to use anything beyond cudaMemcpyDefault
    cudaMemcpy(gpu1Dst_memPtr, host_memPtr, byteSize1, cudaMemcpyDefault)
    cudaMemcpy(gpu2Dst_memPtr, host_memPtr, byteSize2, cudaMemcpyDefault)
    cudaMemcpy(host_memPtr, gpu1Dst_memPtr, byteSize1, cudaMemcpyDefault)
    cudaMemcpy(host_memPtr, gpu2Dst_memPtr, byteSize2, cudaMemcpyDefault)

- UVA support is the enabler for the peer-to-peer (P2P), inter-GPU, data transfer
  - P2P not topic of discussion here
  - UVA is the underpinning technology for P2P
UVA is a Step Forward Relative to Z-C

- Z-C Key Accomplishment: use pointer within device function access host data
  - Z-C focused on a data access issue relevant in the context of functions executed on the device

- UVA had a data access component but also a data transfer component:
  - Data access: A GPU could access data on a different GPU, a novelty back in CUDA 4.0
  - Data transfer: copy data in between GPUs
    - cudaMemcpy is the main character in this play, data transfer initiated on the host side
Zero-Copy, UVA, and How UM Fits In

[1/2]

- Both for Z-C and UVA the memory was allocated on the device with `cudaMalloc` and on the host with `cudaHostAlloc`

- The magic ensued upon the `cudaHostAlloc/cudaMalloc` duo

- Examples of things that can be done:
  - Data on host accessed on the device
  - Data transferred effectively in between devices without intermediate staging on the host
  - Data stored by one GPU accessed directly by a different GPU
  - Etc.
Zero-Copy, UVA, and How UM Fits In

[1/2]

- Unified Memory (UM) eliminates the need to call the cudaMalloc/cudaHostAlloc duo
  - It takes a different perspective on handling memory in the GPU/CPU interplay

- Note that in theory one could get by using Z-C and only calling cudaHostAlloc once. This is not recommended when having repeated accesses by device to host-side memory
  - Each device request that ends up accessing the host-side memory incurs high latency and low bandwidth (relative to the latency and bandwidth of an access to device global memory)

- This is the backdrop against which the role of UM is justified
  - Data is stored and migrated in a user-transparent fashion
    - To the extent possible, the data is right where it’s needed thus enabling fast access
Unified Memory (UM)

- One memory allocation call takes care of memory setup at both ends; i.e., device and host
  - The main actor: the CUDA runtime function cudaMallocManaged()

- New way of perceiving the memory interplay in GPGPU computing
  - No distinction is made between memory on the host and memory on the device
  - It’s just memory, albeit with different access times when accessed by different processors
Unified Memory (UM) – Semantics Issues
[clarifications of terms used on previous slide]

- “processor” (from NVIDIA documentation): any independent execution unit with a dedicated memory management unit (MMU)
  - Includes both CPUs and GPUs of any type and architecture

- “different access time”: time is higher when, for instance, the host accesses for the first time data stored on the device.
  - Subsequent accesses to the same data take place at the bandwidth and latency of accessing host memory
    - This is why access time is different and lower
    - Original access time higher due to migration of data from device to host
  - NOTE: same remarks apply to accesses from the device
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double* aArray, double val, unsigned int sz)
{
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double* mA;
cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));
    for (int i = 0; i < ARRAY_SIZE; i++)
        mA[i] = 1.*i;

double inc_val = 2.0;
increment <<<2, 512 >>>(mA, inc_val, ARRAY_SIZE);
cudaDeviceSynchronize();

double error = 0.;
for (int i = 0; i < ARRAY_SIZE; i++)
    error += fabs(mA[i] - (i + inc_val));

cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;
cudaFree(mA);
return 0;
}
UM vs. Z-C

- Recall that with Z-C, data is always on the host in pinned CPU system memory
  - The device reaches out to it

- UM: data stored on the device but made available where needed
  - Data access and locality managed by underlying system, handling transparent to the user
  - UM provides “single-pointer-to-data” model

- Support for UM called for only *three* additions to CUDA:
  - cudaMallocManaged, __managed__, cudaStreamAttachMemAsync()
Technicalities...

- `cudaError_t cudaMallocManaged ( void** devPtr, size_t size, unsigned int flag)`
  - Returns pointer accessible from both Host and Device
  - Drop-in replacement for `cudaMalloc()` – they are semantically similar
  - Allocates managed memory on the device
    - First two arguments have the expected meaning
  - “flag” controls the default stream association for this allocation
    - `cudaMemAttachGlobal` - memory is accessible from any stream on any device
    - `cudaMemAttachHost` – memory on this device accessible by host only
  - Free memory with the same `cudaFree()`

- `__managed__`
  - Global/file-scope variable annotation combines with `__device__`
  - Declares global-scope migrateable device variable
  - Symbol accessible from both GPU and CPU code

- `cudaStreamAttachMemAsync()`
  - Manages concurrency in multi-threaded CPU applications
UM, Quick Points

- In the current implementation, managed memory is allocated on the device that happens to be active at the time of the allocation.

- Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the `cudaMalloc()` routine.

- All CUDA operations that are valid on device memory are also valid on managed memory.
Example: UM and thrust

```cpp
#include <iostream>
#include <cmath>
#include <thrust/reduce.h>
#include <thrust/system/cuda/execution_policy.h>
#include <thrust/system/omp/execution_policy.h>

const int ARRAY_SIZE = 1000;

int main(int argc, char **argv) {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

    thrust::sequence(mA, mA + ARRAY_SIZE, 1);

    double maximumGPU = thrust::reduce(thrust::cuda::par, mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());
    cudaDeviceSynchronize();
    double maximumCPU = thrust::reduce(thrust::omp::par, mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());

    std::cout << "GPU reduce: " << (std::fabs(maximumGPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;
    std::cout << "CPU reduce: " << (std::fabs(maximumCPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;

    cudaFree(mA);
    return 0;
}
```
Advanced Features: UM

- Managed memory migration is at the page level
  - The default page size is currently the same as the OS page size today (typically 4 KB)

- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCI-E only the dirty pages
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed

- Coherence points are kernel launch and device/stream sync.
  - Important: the same memory cannot be operated upon, at the same time, by the device and host
Advanced Features: UM

- **Issues related to “managed memory size”:**
  - For now, there is no oversubscription of the device memory
    - In fact, if there are several devices available, the max amount of managed memory that can be allocated is the smallest of the memories available on the devices

- **Issues related to “transfer/execution overlap”:**
  - Pages from managed allocations touched by CPU migrated back to GPU before any kernel launch
    - Consequence: there is no kernel execution/data transfer overlap in that stream
    - Overlap possible with UM but just like before it requires multiple kernels in separate streams
      - Enabled by the fact that a managed allocation can be specific to a stream
      - Allows one to control which allocations are synchronized on specific kernel launches, enables concurrency
UM: Coherency Related Issues

- The GPU has *exclusive* access to this memory when any kernel is executed on the device
  - Holds even if the kernel doesn’t touch the managed memory

- The CPU cannot access *any* managed memory allocation or variable as long as GPU is executing

- A cudaDeviceSynchronize() call required for the host to be allowed to access managed memory
  - To this end, any function that logically guarantees the GPU finished execution is acceptable
  - Examples: cudaStreamSynchronize(), cudaMemcpy(), cudaMemcpy(), etc.
__device__ __managed__ int x, y = 2;
__global__ void kernel()
{
    x = 10;
}

int main()
{
    kernel <<< 1, 1 >>>();
    y = 20; // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();
    return 0;
}

__device__ __managed__ int x, y = 2;
__global__ void kernel()
{
    x = 10;
}

int main()
{
    kernel <<< 1, 1 >>>();
    cudaDeviceSynchronize();
    y = 20; // GPU is idle so access is OK
    return 0;
}
UM – Current Limitations in CUDA 6.0

- Ability to allocate more memory than the physically available on the GPU
- Prefetching
- Finer Grain Migration
UM – Why Bother?

1. A matter of convenience
   - Much simpler to write code using this memory model
   - For the casual programmer, the code will run faster due to data locality
     - The runtime will take care of moving the data where it ought to be

2. Looking ahead, physical CPU/GPU integration around the corner – memory will be shared
   - Already the case for integrated GPUs that are part of the system chipset
   - The trend in which the industry is moving (AMD’s APU, Intel’s Haswell, NVIDIA Denver Project)
   - The functionality provided by the current software backend that supports the cudaMallocManaged() paradigm will be eventually implemented in hardware