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

The NVIDIA GPU Memory Ecosystem

Atomic operations in CUDA

The `thrust` library

October 7, 2015

Quote of the Day

“Intelligence is not what we know, but what we do when we don't know.”

-- Jean Piaget (1896-1980)

Before We Get Started

- Issues covered last time:
  - The CUDA memory ecosystem

- Today's topics
  - Wrap up, memory aspects in CUDA
  - Atomic operations
  - GPU parallel computing w/ the `thrust` library [covered by Hammad]

- Assignment:
  - HW04 –due on Oct. 7 at 11:59 PM
  - HW05 – doe on Oct. 13 at 11:59 PM (posted today)

- Midterm Exam: 10/09 (Friday)
  - Review on Th 10/08, at 7:15 PM, room 1153ME
  - Exam is "open everything"
    - Not allowed to communicate with anybody during the exam
    - Having a laptop/table is ok
    - Exam made up of a collection of questions

Global Memory Access Issues

Data Access "Divergence"

- Concept 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 well aligned *and* contiguous block of global memory then the effective bandwidth is reduced
- Useful tip for next slides covered in this "memory access" segment:
  - When you look at a kernel, recall that that's the code that 32 threads; i.e., a warp, execute in lockstep fashion

Global Memory

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The <u>layout of the access</u> to global memory (the pattern of the access)
  - The <u>alignment</u> 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

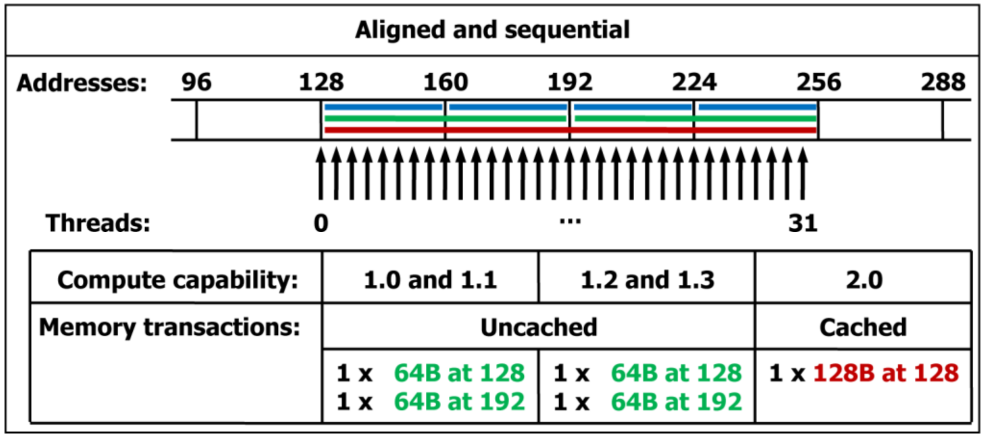
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place

  - 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

Examples of Global Mem. Access by a Warp

- Setup:
  - You want to access `floats` or `integers`
  - In order words, each thread is requesting a 4-Byte word
- Scenario A: access is aligned and sequential

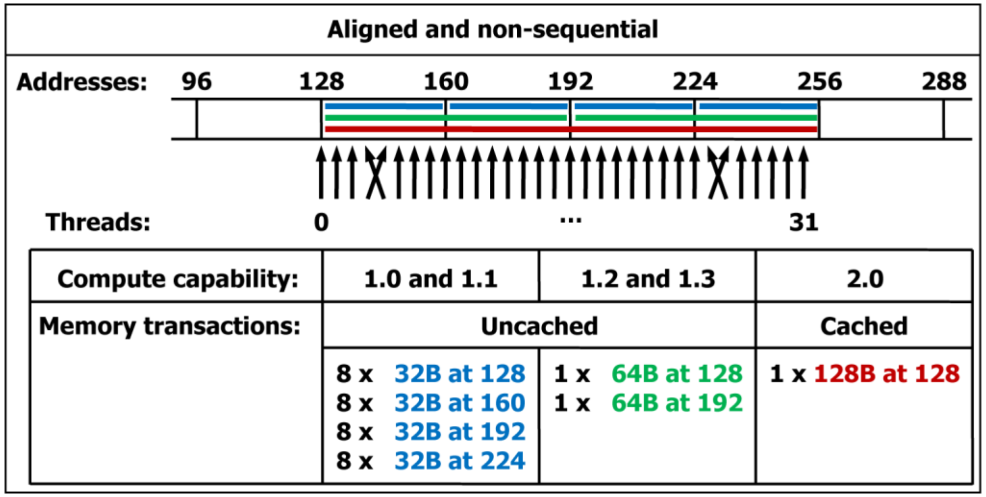


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

- Good to know: any address of memory allocated with **`cudaMalloc`** is a multiple of 256
  - That is, the addressed is 256 byte aligned, which is stronger than 128 byte aligned

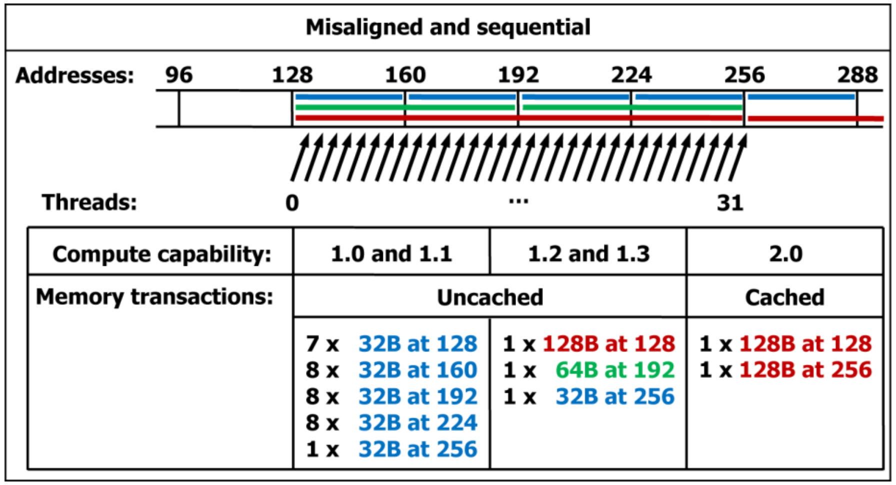
The “Memory Alignment” Issue: How does it manifest”

- Scenario B: Aligned but non-sequential


Aligned and non-sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

- Scenario C: Misaligned and sequential


Misaligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Why is this important?

- Compare Scenario B to Scenario C
- Basically, you have in Scenario C half the effective bandwidth you get in Scenario B
  - Just because of the alignment of your data access
- If your code is memory bound and dominated by this type of access, you might see a doubling of the run time…
- The moral of the story:
  - When you reach out to fetch data from global memory, visualize how a full warp reaches out for access. The important question “Is the access coalesced and well aligned?”
- Scenarios A and B: illustrate what is called a coalesced memory access

Example: Adding Two Matrices

- You have two matrices A and B of dimension NxN (N=32)
- You want to compute C=A+B in parallel
- Code provided below (some details omitted, such as **#define N 32**)

```
// 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

NOTE: On the use of “volatile” w/ Shared Memory in CUDA

- The “volatile” qualifier can be used in C when declaring a variable
  - A way of telling the compiler not to optimize the use of that variable

- Case in point: the `nvcc` (CUDA) compiler might decide to use some unused registers to optimize your code. Example: it’d place variables that you use often in register rather than write them back to shared memory
  - This compiler decision happens without you knowing it
    - In fact you might be thankful to the compiler for optimizing your code

- Imagine `nvcc` optimizes a variables that is of type “shared memory”:
  - Possible scenario: You change this variable during execution, but the change is not reflected back in the shared memory
    - Instead, the register gets the changed value
    - Some other thread might expect to see updated values in shared memory, but this doesn’t happen since what got changed is stored in the register that the compiler pulled into this computation to save time

- Here’s a nice discussion:
  - http://stackoverflow.com/questions/15331009/when-to-use-volatile-with-shared-cuda-memory

Atomic Operations

Choreographing 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 provides a scalable coordination mechanism called atomic memory operation

Race Condition

- A contrived (artificial) example…

```
// 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"

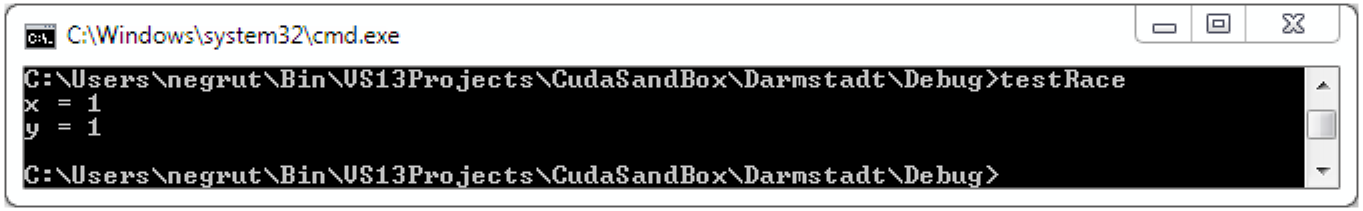
- Handling of an `if-then-else` construct in CUDA
  - First a subset of threads of the warp execute the "`then`" branch
  - Next, the rest of the threads in the warp execute the "`else`" branch
- Question: what happens if in the previous slide if you change the "`if`" to "`if(i>=2)` …" and swap the `then` and the `else` parts?
  - How are things different compared to sequential computing?

Another Example

```
#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\VS13Projects\CudaSandBox\Darmstadt\Debug>testRace
x = 1
y = 1

C:\Users\negrut\Bin\VS13Projects\CudaSandBox\Darmstadt\Debug>
```

Example: Inter-Block Issue

- Would this fly?

```
// 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);
```

Atomics, Introduction

- Atomic memory operations (atomic functions) are used to solve coordination problems in parallel computer systems

- General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.

Atomic Functions
[1/3]

- Atomic functions perform read-modify-write operations on data residing in global and shared memory

```
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = atomicAdd(x, i);    // j is now old value of x;
}

// snippet of code in main.cpp
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

[2/3]

- 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
[3/3]

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates
- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary
- However, none of the atomic updates will be lost
- 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
//
//  color – pointer to picture color data
//  bucket – pointer to histogram buckets, one per color
//

__global__ void histogram(int n, int* color, int* bucket)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n)
  {
    int c = colors[i];
    atomicAdd(&bucket[c], 1);
  }
}
```

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

```
__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 atomic functions for infrequent, sparse, and/or unpredictable global communication
- Attempt to use shared memory and structure algorithms to avoid synchronization whenever possible

Synchronization vs. Coordination

- 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, or of synchronization
    - __syncthreads() establishes a point in the execution of the kernel that every thread in the **block** needs to reach before the execution continues beyond that point

  - The “atomic operation” concept tied to the idea of coordination
    - Threads in a grid of blocks coordinate their execution so that a certain operation invoked in a kernel is conducted in an atomic fashion