CUDA, Further Details

Execution Scheduling
Global Memory
Shared Memory
March 6, 2012

“A computer will do what you tell it to do, but that may be much different from what you had in mind”
Joseph Weizenbaum
Before We Get Started…

- **Last time**
  - Profiling CUDA code to improve performance
  - Examples: debugging and profiling of 1D stencil code
  - Lessons learned:
    - Always spend a bit of time profiling of your code
    - Most likely, what dictates the performance of your code is the efficiency of the memory transactions

- **Today**
  - Scheduling issues, advanced topics
  - Device memory, advanced topics
  - Shared memory, advanced topics

- **Other issues**
  - HW6 due on Th at 11:59 PM
  - Half page proposal for your Midterm Project due on Th
    - For default project (solving dense banded linear system): no need to submit anything.
Acknowledgement

- Several slides in today’s lecture include material provided by James Balfour of NVIDIA
  - A sign such as the one at the bottom of this slides acknowledges his contribution
  - Any inadvertence in these slides belongs to me
Thread Blocks are Executed as Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block can execute independently
Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing
  - Warp scheduling is not something you control through CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead
- Warps with instruction whose inputs are ready are eligible to execute, and will be considered when scheduling
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion

NVIDIA [J. Balfour]→
Filling Warps

- Prefer thread block sizes that result in mostly full warps

**Bad:**  \texttt{kernel<<<N, 1>>>( ... )}  
**Okay:**  \texttt{kernel<<<(N+31) / 32, 32>>>( ... )}  
**Better:**  \texttt{kernel<<<(N+127) / 128, 128>>>( ... )}

- Prefer to have enough threads per block to provide hardware with many warps to switch between

- This is how the GPU hides memory access latency

- Resource like \texttt{__shared__} may constrain number of threads per block

- Algorithm and decomposition will establish some preferred amount of shared data and \texttt{__shared__} allocation
Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

Half the threads in the warp execute the `if` clause, the other half the `else` clause.
The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

Often, this requires that the hardware execute multiple paths through a kernel for a warp.
- For example, both the if clause and the corresponding else clause.
Control Flow Divergence

[3/4]

```
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if (b)
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
```
Control Flow Divergence

[4/4]

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one does need to consider divergence when reasoning about the performance of a program
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here’s an extreme example…

```c
__global__ void dv(int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    switch (i % 32)
    {
    case 0 : x[i] = a(x[i]);
             break;
    case 1 : x[i] = b(x[i]);
             break;
    ...
    case 31: x[i] = v(x[i]);
             break;
    }
}
```
Performance of Divergent Code

Compiler and hardware can detect when all threads in a warp branch in the same direction
  - For example, all take the `if` clause, or all take the `else` clause
  - The hardware is optimized to handle these cases without loss of performance
  - In other words, use of `if` or `switch` does not automatically translate into disaster:

  ```
  if (threadIdx.x / WARP_SIZE >= 2) { }
  ```
  - Creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses
Data Access “Divergence”

- Concept is similar to control divergence and often conflated

- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores

- If a warp doesn’t access a contiguous block of global memory the effective bandwidth is getting reduced

- Remember this: when you look at a kernel you see what a collection of threads; i.e., a warp, is supposed to do in lockstep fashion
The significant change from 1.x to 2.x device capability was the caching of the local and global memory accesses.
Global Memory and Memory Bandwidth

- These change from card to card. On Tesla C1060:
  - 4 GB in GDDR3 RAM
  - Memory clock speed: 800 MHz
  - Memory interface: 512 bits
  - Peak Bandwidth: \(800 \times 10^6 \times (512/8) \times 2 = 102.4 \text{ GB/s}\)

- When reporting effective bandwidth of your application:
  - Formula, effective bandwidth \((B_r \text{ - bytes read, } B_w \text{ - bytes written})\) [measured in GB/s]
    \[
    \text{Effective bandwidth} = \frac{((B_r + B_w)/10^9)}{\text{time}}
    \]
  - Example: kernel copies a \(2048 \times 2048\) matrix from global memory, then copies matrix back to global memory. Does it in a certain amount of \(\text{time}\) [measured in seconds]
    \[
    \text{Effective bandwidth} = \frac{((2048^2 \cdot 4 \cdot 2)/10^9)}{\text{time}}
    \]
  - 4 above comes from four bytes per float, 2 from the fact that the matrix is both read from and written to the global memory. The \(10^9\) used to get an answer in GB/s.
Global Memory

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The layout of the access to global memory (the pattern of the access)
  - The size/alignment of the data you try to fetch from global memory
“Memory Access Layout”

What is it?

- The basic idea:
  - Suppose each thread in a warp accesses a global memory address for a load operation at some point in the execution of the kernel.
  - These threads can access global memory data that is either (a) neatly grouped, or (b) scattered all over the place.
  - Case (a) is called a “coalesced memory access”:
    - If you end up with (b) this will adversely impact the overall program performance.

- Analogy:
  - Can send one truck on six different trips to bring back each time a bundle of wood.
  - Alternatively, can send truck to one place and get it back fully loaded with wood.
Memory Facts, Fermi GPUs

- There is 64 KB of fast memory on each SM that gets split between L1 cache and Shared Memory
  - You can split 64 KB as “L1/Sh: 16/48” or “L1/Sh: 48/16”

- L2 cache: 768 KB – one big pot available to *all* SMs on the device

- L1 and L2 cache used to cache accesses to
  - Local memory, including register spill
  - Global memory

- Whether reads are cached in [L1 & L2] or in [L2 only] can be partially configured on a per-access basis using modifiers to the load or store instruction
Fermi Memory Layout

[credits: NVIDIA]
## GPU-CPU Face Off

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050</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>64* KB L1, per SM 768 KB L2, all SMs 3 GB Device Mem.</td>
<td>- 32 KB L1 cache / core - 256 KB L2 (I&amp;D) cache / core - 8 MB L3 (I&amp;D) shared, all cores</td>
</tr>
<tr>
<td><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>

* - split 48/16
More Memory Facts
[Fermi GPUs]

- All global memory accesses are cached

- A cache line is 128 bytes
  - It maps to a 128-byte aligned segment in device memory

- You can determine at *compile* time (through flags: -dlcm=ca/cg) if you double cache [L1 & L2] or only cache [L2 only]
  - If [L1 & L2], a memory access is serviced with a 128-byte memory transaction
  - If [L2 only], a memory access is serviced with a 32-byte memory transaction
    - This can reduce over-fetch in the case of scattered memory accesses
    - Good for irregular pattern access (sparse linear algebra)
More Memory Facts
[Fermi GPUs]

- If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently.

- The memory access schema is as follows:
  - Two memory requests, one for each half-warp, if the size is 8 bytes.
  - Four memory requests, one for each quarter-warp, if the size is 16 bytes.

- Each memory request is then broken down into cache line requests that are issued independently.

- NOTE: a cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.
When it comes to memory store transactions to global memory:

- First, the L1 cache is invalidated if need be
- Next, the data is stored in L2
- The data is actually written to global memory only if/when the data gets evicted from L2

This strategy works since L2 is visible to all SMs on the device (unlike L1)

How about read-before-write issues?
- Use atomic operations (discussed next lecture)
How to Use L1 and L2

- Should you start programming to leverage L1 and L2 cache?
- The answer is: NO
  - GPU caches are not intended for the same use as CPU caches
    - Smaller sizes (on a per-thread basis, that is), not aimed at temporal reuse
      - Intended to smooth out some access patterns, help with spilled registers, etc.
  - Don’t try to block for L1/L2 like you would on CPU
    - You have 100s to 1000s of run-time scheduled thread hitting the caches
    - Instead of L1, you should start thinking how to leverage Shared Memory
      - Same bandwidth (they *physically* share the same memory)
      - Hardware will not evict behind your back

- Conclusions
  1. Optimize as if no caches were there
  2. The reason why we talk about this: it helps you understand when the GPU is good and when it’s not
A global memory request for a warp is split in two memory requests, one for each half-warp. The following 5-stage protocol is used to determine the memory transactions necessary to service all threads in a half-warp.

**Stage 1:** Find the memory segment that contains the address requested by the lowest numbered active thread. The memory segment size depends on the size of the words accessed by the threads:
- 32 bytes for 1-byte words,
- 64 bytes for 2-byte words,
- 128 bytes for 4-, 8- and 16-byte words.

**Stage 2:** Find all other active threads whose requested address lies in the same segment.

**Stage 3:** Reduce the transaction size, if possible:
- If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
- If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

**Stage 4:** Carry out the transaction and mark the serviced threads as inactive.

**Stage 5:** Repeat until all threads in the half-warp are serviced.
Examples

[Preamble]

- Look at an example that deals with 32 bit words (4 bytes)
- This is the case when handling integers or floats
- Various scenarios are going to be considered to illustrate how the two factors (layout of access & alignment) come into play when accessing global memory
- Note that when handling 32 bit words, “segment size” represents 128 byte data chunks (all aligned at multiples of 128)
  - In what follows, a different color is associated with each 128 byte memory segment
  - In other words, two rows of the same color represent a 128-byte aligned segment
Example: Scenario 1

- Coalesced access in which all threads but one access the corresponding word in a segment.

  ![Diagram showing coalesced access](image)

- This access pattern results in a single 64-byte transaction, indicated by the red rectangle.

- Although one word is not requested, all data in the segment is fetched.
  - Sometimes called an “over-fetch”

- If accesses by threads were permuted within this segment, still one 64-byte transaction would be performed on Tesla C1060.
Example: Scenario 2

- Sequential threads in a half warp access memory that is sequential but not aligned with the segments.

- Given that the addresses fall within a 128-byte segment, a single 128-byte transaction is performed on Tesla C1060.
Example: Scenario 3

- A half warp accesses memory that is sequential but split across two 128-byte segments. Note that the request spans two different memory segments.

- On Tesla C1060, two transactions are performed: one 64-byte transaction and one 32-byte transaction result.
Example: Scenario 4

- Strided access to global memory, as shown in the code snippet below:

```c
__global__ void strideCopy(float *odata, float* idata, int stride) {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- Although a stride of 2 above results in a single transaction, note that half the elements in the transaction are not used and represent wasted bandwidth.
Example: Scenario 4

[Cntd.]

- Strided access to global memory, as shown in the code snippet below:

```c
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- As the stride increases, the effective bandwidth decreases until the point where 16 transactions are issued for the 16 threads in a half warp, as shown in the plot.
Examples of Global Mem. Access by a Warp

- **Setup:**
  - You want to access floats or integers
  - In other words, each thread is requesting a 4-Byte word

- **Scenario A: access is aligned and sequential**

![Diagram showing aligned and sequential access]

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>1 x 64B at 128</td>
<td>1 x 64B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 x 64B at 192</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 x 128B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples of Global Mem. Access by a Warp

- Scenario B: Aligned but non-sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
</tr>
</tbody>
</table>

- Scenario C: Misaligned and sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
</tr>
<tr>
<td></td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 256</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
</tr>
</tbody>
</table>
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 grab data from global memory, visualize how a full warp reaches out for access. Is the access coalesced and well aligned?
Think about this…

- 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
# Technical Specifications and Features

[Short Detour]

This is us: Fermi GPUs on Euler

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65535</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
<td></td>
<td>512</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
<td></td>
<td>512</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td></td>
<td></td>
<td>24</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td></td>
<td></td>
<td>768</td>
<td></td>
<td>1024</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td></td>
<td></td>
<td>8 K</td>
<td></td>
<td>16 K</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
<td></td>
<td>16 KB</td>
<td></td>
<td>48 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td></td>
<td></td>
<td>16 KB</td>
<td></td>
<td>512 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64 KB</td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8 KB</td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 million</td>
</tr>
</tbody>
</table>

Legend:
“multiprocessor” stands for Stream Multiprocessor (what we called SM)