“Once a new technology rolls over you, if you're not part of the steamroller, you're part of the road.”

Stewart Brand
Before We Get Started…

- Last time
  - Wrapped up discussion about execution scheduling on the GPU
  - Discussed global memory access issues in CUDA

- Today
  - Examples, global memory accesses
  - Discuss shared memory accesses in CUDA
  - A couple of comments on HW4

- Other issues
  - HW4 due tonight at 11:59 PM
    - Use Learn@UW drop-box to submit homework
  - HW5 posted, due on March 1, 11:59 PM
  - Please take a look at the latest version of the syllabus, has been updated recently
  - Thursday, Feb. 24
    - TAs Toby Heyn and Arman Pazouki will provide an overview of two Midterm Project topics: Discrete Element Method (DEM) and Collision Detection, respectively
  - Wednesday, Feb 23: no office hours – I will be traveling (leaving on Wd at noon, returning Th evening)
Global Memory Access
Compute Capability 1.3

- 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

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

- Note that even though one word is not requested, all data in the segment are fetched.

- 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.
Looking Beyond Tesla C1060

- Tesla C1060 represents compute capability 1.3. How about other compute capabilities?

- Look at the same example as before
  - Accessing floats or integers for global memory transactions

- Example 1: access is aligned 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>
<tr>
<td>Com. cap:</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>M. trans.</td>
<td>Uncached</td>
<td>Cashed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 192</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Looking Beyond Tesla C1060

[Cntd.]

- Example 2: 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></td>
<td></td>
<td></td>
<td></td>
<td>31</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>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Example 3: 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></td>
<td></td>
<td></td>
<td></td>
<td>31</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>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Think about this…

- Say you use in your program complex data constructs that could be organized using C-structures.

- Based on what we learned 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]

### Compute Capability

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Feature Support**
(Unlisted features are supported for all compute capabilities)

<table>
<thead>
<tr>
<th>Feature Support</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory (Section B.11)</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory (Section B.11)</td>
<td>No</td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp vote functions (Section B.12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__ballot() (Section B.12)</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__threadfence_system() (Section B.5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Reduction with Bank Conflicts
(assume 1024 vector entries stored in shared memory; one block of 512 threads carries out the reduction)
Discuss Shared Memory Issues
You can statically declare shared memory like in the code snippet below:

```c
__global__ void coalescedMultiply(float *a, float* b, float *c,
    int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

- **NOTE:** this makes the variable `aTile` visible to all threads in each block, and only to those threads
- The thread that executes the kernel above sees the `aTile` declaration and understands that all its brother-threads in the block are going to see it too. They will together share this variable
- The same thread, when it sees the variable “row” it understands that it has sole ownership of this variable (variable stored most likely in a register)
Shared Memory
[Tesla C1060]

- Each SM has 16 KB of Shared Memory
  - Physically organized as 16 banks of 4 byte words
  - Note that shared memory can store less data than the registers (16 vs. 64 KB)

- The 16 banks of the Shared Memory are organized like benches in a movie theater
  - You have 256 rows of benches. Each row has 16 benches, in each bench you can “seat” a family of four (bytes). Note that a bank represents a column of benches in the movie theater

- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - All threads in the block have read & write access
Shared Memory: Transaction Rules

- For compute capability 1.x (Newton works with 1.3), the shared memory is organized as 16 banks.
  - Each warp access is split in two → Only half warp accesses shared memory at a time.

- For compute capability 2.x (Fermi), the shared memory is organized as 32 banks.
  - There is no splitting of the warp, all threads in a warp attempt to access shared memory simultaneously.
Q: Is 16K of Shared Memory Enough? Revisit the Matrix Multiplication Example

- One **block** computes one square sub-matrix $C_{sub}$ of size $Block\_Size$

- One **thread** computes one element of $C_{sub}$

- Assume that the dimensions of $A$ and $B$ are multiples of $Block\_Size$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
Matrix Multiplication: Shared Memory Usage

- Each Block requires $2 \times \text{WIDTH}^2 \times 4$ bytes of shared memory storage
  - For WIDTH = 16, each BLOCK requires 2KB, up to 8 Blocks can fit into the Shared Memory of an SM
  - Since each SM can only take 1024 threads, each SM can only take 4 Blocks of 256 threads each
  - Shared memory size is not a limitation for our implementation of the Matrix Multiplication
Shared Memory Architecture

- Common sense observation: in a parallel machine many threads access memory at the same time
  - To service more than one thread, memory is divided into banks
  - Essential to achieve high bandwidth

- Each bank can service one address per cycle
  - The shared memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

- **2-way Bank Conflicts**: Linear addressing stride == 2
- **8-way Bank Conflicts**: Linear addressing stride == 8
Shared Memory Bank Conflicts

- If there are no bank conflicts
  - Shared memory access is as fast as registers
  - Latency is roughly 100x lower than global memory latency

- Share memory access, the fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access an identical address for a fetch operation, there is no bank conflict (broadcast)

- Share memory access, the slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank
How addresses map to banks on Tesla C1060

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- Tesla C1060 has 16 banks
  - Bank you work with = address % 16
  - Same as the number of threads in a half-warp
    - NOTE: There is no such thing as bank conflicts between threads belonging to different half-warps; this issue only relevant for threads from within a single half-warp
Linear Addressing

- Given:
  ```
  __shared__ float sharedM[256];
  float foo = sharedM[baseIndex + s * threadIdx.x];
  ```

- This is bank-conflict-free if s shares no common factors with the number of banks
  - 16 on C1060, so s must be odd
The Math Beyond Bank Conflicts

- We are in a half-warp, and the question is if thread $t_1$ and thread $t_2 > t_1$ might access the same bank of shared memory.
- Let $b$ be the base of the array (the “shareM” pointer on previous slide).
- How should you not choose $s$?

\[
\begin{align*}
&\begin{cases}
    b + st_2 = b + st_1 + 16k, & \text{for some positive integer } k \\
    0 < t_2 - t_1 \leq 15
\end{cases} \\
&\begin{cases}
    16k = s(t_2 - t_1) \\
    0 < t_2 - t_1 \leq 15
\end{cases}
\end{align*}
\]

- If $s=2$, take $k=1$, and then any threads $t_1$ and $t_2$ which are eight apart satisfy the condition above and will have a bank conflict ($[0,8]$, $[1,9]$, etc.) – two way conflict.

- If $s=4$, take $k=2$, any threads $t_1$ and $t_2$ which are four apart will have a bank conflict ($[0,4,8,12]$, $[1,5,9,13]$, etc.) – four way conflict.

- NOTE: you can’t get a bank conflict is $s$ is odd (no quartet $k, s, t_1, t_2$ satisfies the bank conflict condition above). So take stride $s=1,3,5$, etc.
Data types and bank conflicts

- No conflicts below if `shared` is a 32-bit data type:

  ```
  foo = shared[baseIndex + threadIdx.x]
  ```

- But not if the data type is smaller
  - 4-way bank conflicts:
    ```
    __shared__ char shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
  - 2-way bank conflicts:
    ```
    __shared__ short shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are struct members:

  ```c
  struct vector { float x, y, z; }
  struct myType {
      float f;
      int c;
  }
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for vector; struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)

  ```c
  struct vector v = vectors[baseIndex + threadIdx.x];
  ```

- This has 2-way bank conflicts for myType; (2 accesses per thread)

  ```c
  struct myType m = myTypes[baseIndex + threadIdx.x];
  ```
Common Array Bank Conflict Patterns 1D

- Each thread loads 2 elements into shared memory:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```c
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
  - Not in shared memory usage where there is no cache line effects but banking effects
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of blockDim elements.

\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}];
\]
\[
\text{shared}[\text{tid} + \text{blockDim}.x] = \text{global}[\text{tid} + \text{blockDim}.x];
\]
Vector Reduction **without** Bank Conflicts
(assume 1024 vector entries stored in shared memory; one block of 512 threads carries out the reduction)