CSCI-GA.3033-012

Graphics Processing Units (GPUs): Architecture and Programming

Lecture 6: CUDA Memories

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
Let’s Start With An Example

- G80 supports 86.4 GB/s of global memory access
- Single precision floating point = 4 bytes
- Then we cannot load more than \( \frac{86.4}{4} = 21.6 \text{ giga single precision data per second} \)
- Theoretical peak performance of G80 is 367 gigaglops!
Main Goals for This Lecture

- How to make the best use of the GPU memory system?
- How to deal with hardware limitation?
- Fastest.
- Only accessible by a thread.
- Lifetime of a thread
Shared Memory
- Extremely fast
- Highly parallel
- Restricted to a block
- Example: Fermi’s shared/L1 is 1+TB/s aggregate
Global Memory

- Typically implemented in DRAM
- High access latency: 400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/s

Traffic congestion prevents all but a few threads from making progress.
Constant Memory
- Read only
- Short latency and high bandwidth when all threads access the same location
**local memory**

Does not physically exist. It is an abstraction to the local scope of a thread. Actually put in global memory by the compiler.
The variable must be declared within the kernel function body; and will be available only within the kernel code.

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic variables other than arrays</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>shared</strong>, int SharedVar;</td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, int GlobalVar;</td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>constant</strong>, int ConstVar;</td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td>Variable Declaration</td>
<td>Memory</td>
<td>Scope</td>
<td>Lifetime</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>----------</td>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>Automatic variables other than arrays</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>shared</strong>, int SharedVar;</td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, int GlobalVar;</td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>constant</strong>, int ConstVar;</td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>
• Declaration of constant variables must be outside any function body.
• Currently total size of constant variables in an application is limited to 64KB.

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic variables other than arrays</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>shared</strong>, int SharedVar;</td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, int GlobalVar;</td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>constant</strong>, int ConstVar;</td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>
Where to Declare Variables?

Can host access it?

Yes
- Global
- Constant

No
- Register (automatic)
- Shared
- Local

Outside of any Function

In the kernel
Computation vs Memory Access

- Compute to global memory access (CGMA) ratio
- The number of FP calculations performed for each access to the global memory within a region in a CUDA program.
Computation vs Memory Access

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
```

2 memory accesses
1 FP multiplication
1 FP addition
so  CGMA = 1
Reducing Global Memory Traffic

• Global memory access is performance bottleneck.
• The lower CGMA the lower the performance
• Reducing global memory access enhances performance.
• A common strategy is tiling: partition the data into subsets called tiles, such that each tile fits into the shared memory.
Back to Matrix Multiplication
# Back to Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>(P_{0,0}) thread_{0,0}</th>
<th>(P_{1,0}) thread_{1,0}</th>
<th>(P_{0,1}) thread_{0,1}</th>
<th>(P_{1,1}) thread_{1,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(M_{0,0} \times N_{0,0})</td>
<td>(M_{0,0} \times N_{1,0})</td>
<td>(M_{0,1} \times N_{0,0})</td>
<td>(M_{0,1} \times N_{1,0})</td>
<td></td>
</tr>
<tr>
<td>(M_{1,0} \times N_{0,1})</td>
<td>(M_{1,0} \times N_{1,1})</td>
<td>(M_{1,1} \times N_{0,1})</td>
<td>(M_{1,1} \times N_{1,1})</td>
<td></td>
</tr>
<tr>
<td>(M_{2,0} \times N_{0,2})</td>
<td>(M_{2,0} \times N_{1,2})</td>
<td>(M_{2,1} \times N_{0,2})</td>
<td>(M_{2,1} \times N_{1,2})</td>
<td></td>
</tr>
<tr>
<td>(M_{3,0} \times N_{0,3})</td>
<td>(M_{3,0} \times N_{1,3})</td>
<td>(M_{3,1} \times N_{0,3})</td>
<td>(M_{3,1} \times N_{1,3})</td>
<td></td>
</tr>
</tbody>
</table>

Access order
Back to Matrix Multiplication

• The basic idea is to make threads that use common elements collaborate.
• Each thread can load different elements into the shared memory before calculations.
• These elements will be used by the thread that loaded them and other threads that share them.
# Back to Matrix Multiplication

<table>
<thead>
<tr>
<th>Time</th>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{0,0}$</td>
<td>$\begin{bmatrix} Md_{0,0} \ Mds_{0,0} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Md_{2,0} \ Mds_{0,0} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{bmatrix} Nd_{0,0} \ Nds_{0,0} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Nd_{0,2} \ Nds_{0,0} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$PValue_{0,0} = Mds_{0,0} \cdot Nds_{0,0} + Mds_{1,0} \cdot Nds_{0,1}$</td>
<td>$PValue_{0,0} = Mds_{0,0} \cdot Nds_{0,0} + Mds_{1,0} \cdot Nds_{0,1}$</td>
</tr>
<tr>
<td>$T_{1,0}$</td>
<td>$\begin{bmatrix} Md_{1,0} \ Mds_{1,0} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Md_{3,0} \ Mds_{1,0} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{bmatrix} Nd_{1,0} \ Nds_{1,0} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Nd_{1,2} \ Nds_{1,0} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$PValue_{1,0} = Mds_{0,0} \cdot Nds_{1,0} + Mds_{1,0} \cdot Nds_{1,1}$</td>
<td>$PValue_{1,0} = Mds_{0,0} \cdot Nds_{1,0} + Mds_{1,0} \cdot Nds_{1,1}$</td>
</tr>
<tr>
<td>$T_{0,1}$</td>
<td>$\begin{bmatrix} Md_{0,1} \ Mds_{0,1} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Md_{2,1} \ Mds_{0,1} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{bmatrix} Nd_{0,1} \ Nds_{0,1} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Nd_{0,3} \ Nds_{0,1} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$PdValue_{0,1} = Mds_{0,1} \cdot Nds_{0,0} + Mds_{1,1} \cdot Nds_{0,1}$</td>
<td>$PdValue_{0,1} = Mds_{0,1} \cdot Nds_{0,0} + Mds_{1,1} \cdot Nds_{0,1}$</td>
</tr>
<tr>
<td>$T_{1,1}$</td>
<td>$\begin{bmatrix} Md_{1,1} \ Mds_{1,1} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Md_{3,1} \ Mds_{1,1} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{bmatrix} Nd_{1,1} \ Nds_{1,1} \end{bmatrix}$</td>
<td>$\begin{bmatrix} Nd_{1,3} \ Nds_{1,1} \end{bmatrix}$</td>
</tr>
<tr>
<td></td>
<td>$PdValue_{1,1} = Mds_{0,1} \cdot Nds_{1,0} + Mds_{1,1} \cdot Nds_{1,1}$</td>
<td>$PdValue_{1,1} = Mds_{0,1} \cdot Nds_{1,0} + Mds_{1,1} \cdot Nds_{1,1}$</td>
</tr>
</tbody>
</table>
Potential reduction in global memory traffic in matrix multiplication example is proportional to the dimension of the blocks used.

- With $N \times N$ blocks the potential reduction would be $N$

If an input matrix is of dimension $M$ and the tile size is $TILE\_WIDTH$, the dot product will be performed in $M/TILE\_WIDTH$ phases.
Back to Matrix Multiplication
Back to Matrix Multiplication

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) 
{
  1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  3. int bx = blockIdx.x; int by = blockIdx.y;
  4. int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the Pd element to work on
  5. int Row = by * TILE_WIDTH + ty;
  6. int Col = bx * TILE_WIDTH + tx;

  7. float Pvalue = 0;
  // Loop over the Md and Nd tiles required to compute the Pd element
  8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {

    // Collaborative loading of Md and Nd tiles into shared memory
    9.   Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
  10.   Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
  11.   __syncthreads();

    12. for (int k = 0; k < TILE_WIDTH; ++k)
  13.     Pvalue += Mds[ty][k] * Nds[k][tx];
  14.     __syncthreads();

  15. Pd[Row*Width + Col] = Pvalue;
}
```

The Phases
```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
  }
  Pd[Row*Width + Col] = Pvalue;
}
```

to be sure needed elements are loaded

to be sure calculations are completed
Exercise

Can we use shared memory to reduce global memory bandwidth for matrix addition?
Do you Remember the G80 example?

• 86.4 GB/s global memory bandwidth

• In matrix multiplication if we use 16x16 tiles -> reduction in memory traffic by a factor of 16

• Global memory can now support

  \[(86.4/4) \times 16\] = 345.6 gigaflops -> very close to the peak (367 gigaflops).
Memory As Limiting Factor to Parallelism

• Limited CUDA memory limits the number of threads that can execute simultaneously in SM for a given application
  – The more memory location each thread requires, the fewer the number of threads per SM
Memory As Limiting Factor to Parallelism

- Example: Registers
  - G80 has 8K registers per SM -> 128K registers for entire processor.
  - G80 can accommodate up to 768 threads per SM
  - To fill this capacity each thread can use only 8K/768 = 10 registers.
  - If each thread uses 11 registers -> threads per SM are reduced -> per block granularity
  - e.g. if block contains 256 threads the number of threads will be reduced by 256 -> lowering the number of threads/SM from 768 to 512 (i.e. 1/3 reduction of threads!)
Memory As Limiting Factor to Parallelism

• Example: Shared memory
  – G80 has 16KB of shared memory per SM
  – SM accommodates up to 8 blocks
  – To reach this maximum each block must not exceed $16\text{KB}/8 = 2\text{KB}$ of memory.
  – e.g. if each block uses 5KB -> no more than 3 blocks can be assigned to each SM
GPU Compute Capability

- A way to express hardware resources in a standardized form
- Starts with `compute 1.0`
- Higher level compute capability defines a superset of features of those at lower levels
## GPU Compute Capability

### Compute 1.0

<table>
<thead>
<tr>
<th>Features</th>
<th>Compute 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of stream processors per SM</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum grid dimension ((x, y))</td>
<td>65,535, 65,535</td>
</tr>
<tr>
<td>Maximum block dimension ((x, y, z))</td>
<td>512, 512, 64</td>
</tr>
<tr>
<td>Threads in a warp</td>
<td>32</td>
</tr>
<tr>
<td>Registers per SM</td>
<td>8192 (8 K)</td>
</tr>
<tr>
<td>Shared memory per SM</td>
<td>16,384 (16 K)</td>
</tr>
<tr>
<td>Banks in shared memory</td>
<td>16</td>
</tr>
<tr>
<td>Total constant memory</td>
<td>65,536 (64 K)</td>
</tr>
<tr>
<td>Cache working set for constants per SM</td>
<td>8192 (8 K)</td>
</tr>
<tr>
<td>Local memory per thread</td>
<td>16,384 (16 K)</td>
</tr>
<tr>
<td>Cache working set for texture per SM</td>
<td>6 to 8 kB</td>
</tr>
<tr>
<td>Maximum number of active blocks per SM</td>
<td>8</td>
</tr>
<tr>
<td>Maximum active warps per SM</td>
<td>24</td>
</tr>
<tr>
<td>Maximum active threads per SM</td>
<td>768</td>
</tr>
<tr>
<td>1D texture bound to CUDA array—maximum width</td>
<td>2^{13}</td>
</tr>
<tr>
<td>1D texture bound to linear memory—maximum width</td>
<td>2^{27}</td>
</tr>
<tr>
<td>2D texture bound to linear memory or CUDA array—maximum dimension ((x, y))</td>
<td>2^{16}, 2^{15}</td>
</tr>
<tr>
<td>3D texture bound to a CUDA array—maximum dimension ((x, y, z))</td>
<td>2^{11}, 2^{11}, 2^{11}</td>
</tr>
<tr>
<td>Maximum kernel size</td>
<td>2 million microcode instructions</td>
</tr>
</tbody>
</table>
Conclusions

• Using memory effectively will likely require the redesign of the algorithm.
• The ability to reason about hardware limitations when developing an application is a key concept of computational thinking.
• We are done with chp 6.