PSU CS 410/510 General Purpose GPU Computing
Prof. Karen L. Karavanic
Summer 2014

The CUDA Memory Model and Locality

Includes materials © DavKirk/NVIDIA id and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread registers (~1 cycle)
  - Read/write per-block shared memory (~5 cycles)
  - Read/write per-grid global memory (~500 cycles)
  - Read/only per-grid constant memory (~5 cycles with caching)
## CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</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 ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __shared__, or __constant__

- Automatic variables without any qualifier reside in a register
  - Except per-thread arrays that reside in global memory
Device Shared Memory

- Will ds_M and ds_N be visible to all threads in: grid? Block?

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
}
```

- Are ds_M and ds_N read only?

- Are ds_M and ds_N read only?
Using Shared Memory with Tiling

- **Global memory** resides in device memory (DRAM)
  - slow access, limited bandwidth
- We can take advantage of fast shared memory by **tiling** input data:
  - **Partition** data into subsets that fit into shared memory
  - **Handle** each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory
Matrix-Matrix Multiplication using Shared Memory
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column idenx 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 += d_M[Row*Width+k]* d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
}
Matrix Multiplication Performance
ex: Fermi

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 4*1,000 = 4,000 GB/s required to achieve peak FLOP rating
  - 150 GB/s limits the code at 37.5 GFLOPS

- The actual code runs at about 25 GFLOPS

- Need to drastically cut down memory accesses to get closer to the peak 1,000 GFLOPS
Shared Memory Blocking Basic Idea

Global Memory

Thread 1

Thread 2

Global Memory

On-chip Memory

Thread 1

Thread 2

...
Tiling: Outline of Technique

- Identify a block/tile of global memory content that is accessed by multiple threads
- Load the block/tile from global memory into shared memory
- Have the multiple threads access their data directly from the shared (on-chip) memory
- Move on to the next block/tile
- [Note: synchronization alert!]
Idea: Use Shared Memory to reuse global memory data

- Each input element is read by WIDTH threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - Tiled algorithms
Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd.
Loading a Tile

• All threads in a block participate
  – Each thread loads one Md element and one Nd element in based tiled code

• Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).
Work for Block (0,0)
Work for Block (0,0)

<table>
<thead>
<tr>
<th>N_{0,0}</th>
<th>N_{0,1}</th>
<th>N_{0,2}</th>
<th>N_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{1,0}</td>
<td>N_{1,1}</td>
<td>N_{1,2}</td>
<td>N_{1,3}</td>
</tr>
<tr>
<td>N_{2,0}</td>
<td>N_{2,1}</td>
<td>N_{2,2}</td>
<td>N_{2,3}</td>
</tr>
<tr>
<td>N_{3,0}</td>
<td>N_{3,1}</td>
<td>N_{3,2}</td>
<td>N_{3,3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M_{0,0}</th>
<th>M_{0,1}</th>
<th>M_{0,2}</th>
<th>M_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{1,0}</td>
<td>M_{1,1}</td>
<td>M_{1,2}</td>
<td>M_{1,3}</td>
</tr>
<tr>
<td>M_{2,0}</td>
<td>M_{2,1}</td>
<td>M_{2,2}</td>
<td>M_{2,3}</td>
</tr>
<tr>
<td>M_{3,0}</td>
<td>M_{3,1}</td>
<td>M_{3,2}</td>
<td>M_{3,3}</td>
</tr>
</tbody>
</table>
Work for Block (0,0)
## Work for Block (0,0)

### Matrix Representation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{0,0}$</td>
<td>$N_{0,1}$</td>
<td>$N_{0,2}$</td>
<td>$N_{0,3}$</td>
<td></td>
</tr>
<tr>
<td>$N_{1,0}$</td>
<td>$N_{1,1}$</td>
<td>$N_{1,2}$</td>
<td>$N_{1,3}$</td>
<td></td>
</tr>
<tr>
<td>$N_{2,0}$</td>
<td>$N_{2,1}$</td>
<td>$N_{2,2}$</td>
<td>$N_{2,3}$</td>
<td></td>
</tr>
<tr>
<td>$N_{3,0}$</td>
<td>$N_{3,1}$</td>
<td>$N_{3,2}$</td>
<td>$N_{3,3}$</td>
<td></td>
</tr>
</tbody>
</table>

### Matrix Representation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{0,0}$</td>
<td>$M_{0,1}$</td>
<td>$M_{0,2}$</td>
<td>$M_{0,3}$</td>
<td></td>
</tr>
<tr>
<td>$M_{1,0}$</td>
<td>$M_{1,1}$</td>
<td>$M_{1,2}$</td>
<td>$M_{1,3}$</td>
<td></td>
</tr>
<tr>
<td>$M_{2,0}$</td>
<td>$M_{2,1}$</td>
<td>$M_{2,2}$</td>
<td>$M_{2,3}$</td>
<td></td>
</tr>
<tr>
<td>$M_{3,0}$</td>
<td>$M_{3,1}$</td>
<td>$M_{3,2}$</td>
<td>$M_{3,3}$</td>
<td></td>
</tr>
</tbody>
</table>

### Matrix Representation

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{0,0}$</td>
<td>$P_{0,1}$</td>
<td>$P_{0,2}$</td>
<td>$P_{0,3}$</td>
<td></td>
</tr>
<tr>
<td>$P_{1,0}$</td>
<td>$P_{1,1}$</td>
<td>$P_{1,2}$</td>
<td>$P_{1,3}$</td>
<td></td>
</tr>
<tr>
<td>$P_{2,0}$</td>
<td>$P_{2,1}$</td>
<td>$P_{2,2}$</td>
<td>$P_{2,3}$</td>
<td></td>
</tr>
<tr>
<td>$P_{3,0}$</td>
<td>$P_{3,1}$</td>
<td>$P_{3,2}$</td>
<td>$P_{3,3}$</td>
<td></td>
</tr>
</tbody>
</table>

### Connections

- $N_{0,0} \rightarrow N_{0,1}$
- $N_{1,0} \rightarrow N_{1,1}$
- $M_{0,0} \rightarrow M_{0,1}$
- $M_{1,0} \rightarrow M_{1,1}$

**SM**
Work for Block (0,0)
Loading an Input Tile

Accessing tile 0 2D indexing:
M[Row][tx]
N[ty][Col]
Loading an Input Tile

Accessing tile 1 in 2D indexing:
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]
Loading Input Tile m

However, M and N are dynamically allocated and can only use 1D indexing:

\[
\begin{align*}
M[Row][m*\text{TILE WIDTH}+tx] \\
M[Row*\text{Width} + m*\text{TILE WIDTH} + tx]
\end{align*}
\]

\[
\begin{align*}
N[m*\text{TILE WIDTH}+ty][Col] \\
N[(m*\text{TILE WIDTH}+ty) * \text{Width} + Col]
\end{align*}
\]
Barrier Synchronization in CUDA

- __syncthreads()

- All threads in the same block must reach the __syncthreads() before any can move on

- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed
Figure 4.11 An example execution timing of barrier synchronization.
Tiled Matrix Multiplication Kernel

__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[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
        ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
        ds_N[ty][tx] = d_N[(m*TILE_WIDTH+ty)*Width+Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += ds_M[ty][k] * ds_N[k][tx];
        __syncthreads();
    }
    d_P[Row*Width+Col] = Pvalue;
}
Notes on Tiled MM Kernel

• 1&2: will hold one tile for each input matrix at a time
• 3&4: just convenience; different for each thread
• 8: why Width / TILE_WIDTH ?
• 14: Should __syncthreads be inside the for loop?
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, 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 += d_M[Row*Width+k] * d_N[k*Width+Col];

    d_P[Row*Width+Col] = Pvalue;
}
Shared Memory and Threading

• Each SM in Fermi has 16KB or 48KB shared memory*
  – Shared memory size is implementation dependent!
  – For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  – Can potentially have up to 8 Thread Blocks actively executing
    • This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  – The next TILE_WIDTH 32 would lead to 2*32*32*4B = 8KB shared memory usage per thread block, allowing 2 or 6 thread blocks active at the same time

• Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  – The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!

*Configurable vs L1, total 64KB
Device Query

• Number of devices in the system
  
  ```
  int dev_count;
  cudaGetDeviceCount( &dev_count);
  ```

• Capability of devices
  
  ```
  cudaDeviceProp       dev_prop;
  for (i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties( &dev_prop, i);
    // decide if device has sufficient resources and capabilities
  }
  ```

• `cudaDeviceProp` is a built-in C structure type
  
  – `dev_prop.dev_prop.maxThreadsPerBlock`
  – `Dev_prop.sharedMemoryPerBlock`
  – …