VSCSE Summer School 2009

Many-core Processors for Science and Engineering Applications

Lecture 4: CUDA Memories
G80 Implementation of CUDA Memories

Each thread can:
- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read/only per-grid constant memory
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><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local</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 __local__, __shared__, or __constant__

- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory
Where to Declare Variables?

Can host access it?

- global constant (yes)
- register (automatic)
- shared local (no)

Outside of any Function

In the kernel
Variable Type Restrictions

- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    ```c
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable:
    ```c
    float* ptr = &GlobalVar;
    ```
A Common Programming Strategy

• Global memory resides in device memory (DRAM) - much slower access than shared memory
• So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  – 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
A Common Programming Strategy (Cont.)

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

For texture memory usage, see courses.ece.uiuc.edu/ece498/al.
GPU Atomic Integer Operations

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

- Requires hardware with compute capability 1.1
Matrix Multiplication using Shared Memory
Revised Matrix Multiplication Kernel using Multiple Blocks

```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][k] * Nd[k][Col];
    }

    Pd[Row][Col] = Pvalue;
}
```
How about performance on G80?

• 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
  – \(4 \times 346.5 = 1386\) GB/s required to achieve peak FLOP rating
  – 86.4 GB/s limits the code at 21.6 GFLOPS

• The actual code runs at about 15 GFLOPS

• Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS
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
A Small Example
Every M and N Element is used exactly twice in generating a 2X2 tile of P

<table>
<thead>
<tr>
<th>P&lt;sub&gt;0,0&lt;/sub&gt;</th>
<th>P&lt;sub&gt;1,0&lt;/sub&gt;</th>
<th>P&lt;sub&gt;0,1&lt;/sub&gt;</th>
<th>P&lt;sub&gt;1,1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>thread&lt;sub&gt;1,0&lt;/sub&gt;</td>
<td>thread&lt;sub&gt;0,1&lt;/sub&gt;</td>
<td>thread&lt;sub&gt;1,1&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;0,0&lt;/sub&gt; * N&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>M&lt;sub&gt;0,0&lt;/sub&gt; * N&lt;sub&gt;1,0&lt;/sub&gt;</td>
<td>M&lt;sub&gt;0,1&lt;/sub&gt; * N&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>M&lt;sub&gt;0,1&lt;/sub&gt; * N&lt;sub&gt;1,0&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;1,0&lt;/sub&gt; * N&lt;sub&gt;0,1&lt;/sub&gt;</td>
<td>M&lt;sub&gt;1,0&lt;/sub&gt; * N&lt;sub&gt;1,1&lt;/sub&gt;</td>
<td>M&lt;sub&gt;1,1&lt;/sub&gt; * N&lt;sub&gt;0,1&lt;/sub&gt;</td>
<td>M&lt;sub&gt;1,1&lt;/sub&gt; * N&lt;sub&gt;1,1&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;2,0&lt;/sub&gt; * N&lt;sub&gt;0,2&lt;/sub&gt;</td>
<td>M&lt;sub&gt;2,0&lt;/sub&gt; * N&lt;sub&gt;1,2&lt;/sub&gt;</td>
<td>M&lt;sub&gt;2,1&lt;/sub&gt; * N&lt;sub&gt;0,2&lt;/sub&gt;</td>
<td>M&lt;sub&gt;2,1&lt;/sub&gt; * N&lt;sub&gt;1,2&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;3,0&lt;/sub&gt; * N&lt;sub&gt;0,3&lt;/sub&gt;</td>
<td>M&lt;sub&gt;3,0&lt;/sub&gt; * N&lt;sub&gt;1,3&lt;/sub&gt;</td>
<td>M&lt;sub&gt;3,1&lt;/sub&gt; * N&lt;sub&gt;0,3&lt;/sub&gt;</td>
<td>M&lt;sub&gt;3,1&lt;/sub&gt; * N&lt;sub&gt;1,3&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Access order
Breaking Md and Nd into Tiles

<table>
<thead>
<tr>
<th>Md_{0,0}</th>
<th>Md_{1,0}</th>
<th>Md_{2,0}</th>
<th>Md_{3,0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pd_{0,0}</td>
<td>Pd_{1,0}</td>
<td>Pd_{2,0}</td>
<td>Pd_{3,0}</td>
</tr>
<tr>
<td>Pd_{0,1}</td>
<td>Pd_{1,1}</td>
<td>Pd_{2,1}</td>
<td>Pd_{3,1}</td>
</tr>
<tr>
<td>Pd_{0,2}</td>
<td>Pd_{1,2}</td>
<td>Pd_{2,2}</td>
<td>Pd_{3,2}</td>
</tr>
<tr>
<td>Pd_{0,3}</td>
<td>Pd_{1,3}</td>
<td>Pd_{2,3}</td>
<td>Pd_{3,3}</td>
</tr>
<tr>
<td>Nd_{0,0}</td>
<td>Nd_{1,0}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nd_{0,1}</td>
<td>Nd_{1,1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nd_{0,2}</td>
<td>Nd_{1,2}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nd_{0,3}</td>
<td>Nd_{1,3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Each phase uses one tile from Md and one from Nd

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{0,0}</td>
<td>M_{d,0,0}</td>
</tr>
<tr>
<td>↓ M_{d,0,0}</td>
<td>N_{d,0,0}</td>
</tr>
<tr>
<td>T_{1,0}</td>
<td>M_{d,1,0}</td>
</tr>
<tr>
<td>↓ M_{d,1,0}</td>
<td>N_{d,1,0}</td>
</tr>
<tr>
<td>T_{0,1}</td>
<td>M_{d,0,1}</td>
</tr>
<tr>
<td>↓ M_{d,0,1}</td>
<td>N_{d,0,1}</td>
</tr>
<tr>
<td>T_{1,1}</td>
<td>M_{d,1,1}</td>
</tr>
<tr>
<td>↓ M_{d,1,1}</td>
<td>N_{d,1,1}</td>
</tr>
</tbody>
</table>

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009
First-order Size Considerations in G80

• Each **thread block** should have many threads
  – TILE_WIDTH of 16 gives 16*16 = 256 threads

• There should be many thread blocks
  – A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks

• Each thread block perform $2 \times 256 = 512$ float loads from global memory for $256 \times (2 \times 16) = 8,192$ mul/add operations.
  – Memory bandwidth no longer a limiting factor
CUDA Code – Kernel Execution Configuration

// Setup the execution configuration

dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH);
Tiled Matrix Multiplication Kernel

```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[tx][ty] = Md[(m*TILE_WIDTH + tx)*Width+Row];
        Nds[tx][ty] = Nd[Col*Width+(m*TILE_WIDTH + ty)];
        __syncthreads();

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

- Each **block** computes one square sub-matrix $P_{d_{sub}}$ of size `TILE_WIDTH`
- Each **thread** computes one element of $P_{d_{sub}}$
G80 Shared Memory and Threading

• Each SM in G80 has 16KB shared memory
  – SM size is implementation dependent!
  – For TILE_WIDTH = 16, each thread block uses $2 \times 256 \times 4B = 2KB$ of shared memory.
  – Can potentially have up to 8 Thread Blocks actively executing
    • This allows up to $8 \times 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
    – The next TILE_WIDTH 32 would lead to $2 \times 32 \times 32 \times 4B = 8KB$ shared memory usage per thread block, allowing only up to two thread blocks active at the same time

• Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  – The 86.4B/s bandwidth can now support $(86.4/4) \times 16 = 347.6$ GFLOPS!
Tiling Size Effects

- not tiled
- tiled only
- tiled & unrolled
- 4x4 tiles
- 8x8 tiles
- 12x12 tiles
- 16x16 tiles
- tiled only
- tiled & unrolled

GFLOPS
Summary - Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__,... __global__, __constant__, __texture__

- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)

- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlbIVarPtr, bytes)
  - transfer data from host to device – cudaMemcpy(d_GlbIVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>( args… );
  - transfer results from device to host – cudaMemcpy(h_GlbIVarPtr,…)
  - optional: compare against golden (host computed) solution

- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
    - syncthreads()…

- Other functions
  - float handyFunction(int inVar…);