ECE 8823A
GPU Architectures

Module 4: Memory Model and Locality

© David Kirk/NVIDIA and Wen-mei Hwu, 2007-2012
ECE408/CS483/ECE498al, University of Illinois, Urbana-Champaign

Reading Assignment

- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 5
- CUDA Programming Guide
Objective

• To understand the different memory spaces in the CUDA programming model

• To learn to efficiently use the important levels of the CUDA memory hierarchy
  – Registers, shared memory, global memory
  – Tiled algorithms and barrier synchronization

Execution View of the Memory Hierarchy

• Memory interference reduces memory bandwidth utilization

• Low ops/byte ratio reduces effectiveness of memory bandwidth
  – Leads to low utilization of compute capacity

• Need a richer memory hierarchy to help optimize performance

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)

Impact of Memory Hierarchies

- Another reason for memory hierarchies
  \[ \text{Perf} \left( \frac{\text{ops}}{S} \right) = \text{Power} (W) \times \text{Efficiency} \left( \frac{\text{ops}}{\text{joule}} \right) \]
Automatic Variables

- Private version created for each thread

Matrix Multiplication Revisited

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
        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;
    }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Shared Memory

- Accessible by all threads in a block
- Parallel access

Shared Memory in CUDA

- A special type of memory whose contents are explicitly declared and used in the source code
  - Located in the processor
  - Accessed at much higher speed (in both latency and throughput)
  - Still accessed by memory access instructions
  - Commonly referred to as scratchpad memory in computer architecture
  - Shared by threads in a block
- Algorithmic optimizations to use shared memory
Global Memory

Means for a thread to collaborate across thread blocks
- Note: no synchronization between thread blocks
- Availability of atomics and fences (used in some circumstances)

Constant Memory

- Declaration of constants across all threads
- Cached for faster access
- High BW read-only access
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
- Note programmer controlled placement

Use of Constants: 1D Convolution

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ghost nodes
0 1 2 3 4

P[]

N[]

M[] (constant array)

1. __global__ void convolution_1D_basic_kernel(float *N, float *M, float *P, int Mask_Width, int Width) {
2. int i = blockIdx.x*blockDim.x + threadIdx;
3. float Pvalue = 0;
4. int N_start = i - (Mask_Width/2);
5. For (int j = 0; j < Width; j++) {
6. if (N_start +j >=0 && N_start + j< Width){
7. Pvalue += N[N_start +j] * M[j];
8. }
9. P[i] = Pvalue;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al 2007-2012
Use of Constant Memory

- Consider
  - Size of \( M[] \) is typically small
  - \( M[] \) is constant
  - All threads access the same elements at the same time

Management

- Constant memory must be explicitly managed
  - Allocation: constants are treated as global variables
  - Copying: copied into global memory and can be cached into a separate, dedicated cache
- Kernel functions access constants as global variables
  - Cached in the constant cache
  - Broadcast capability to all threads in a warp
### Modified: 1D Convolution

```c
__global__ void convolution_1D_basics_kernel(float *N, float *P, int Mask_Width, int Width) {
    int i = blockIdx.x*blockDim.x + threadIdx;
    float Pvalue = 0;
    int N_start = i – (Mask_Width/2);
    For (int j= 0; j <Width; j++) {
        if (N_start +j >=0 && N_start + j< Width){
            Pvalue += N[N_start +j] * M[j];
        }
    }
    P[i] = Pvalue;
}
```

- `N[]`
- `M[]` (constant array)
- `P[]` (ghost nodes)
- Declare as constant
- `__constant__ int tempM[Mask_Width]`
- `M[]` not passed in as a parameter
- Accessed as a global array
- No need to allocate on device
- Transfer using different API function `cudaMemcpyToSymbol(M, tempM, size)`

### Refining the Memory Hierarchy

- For best performance, index into constant cache should not be a function of thread ID!

- Separate **read-only cache** that the compiler uses – distinct from from the constant cache
  - Accessed with additional qualifier (**__restrict__**)

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Using Shared Memory

- Global memory resides in device memory (DRAM) - slow access
- So, a profitable way of performing computation on the device is to tile input 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

```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];
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Matrix-Matrix Multiplication using Shared Memory

Base Matrix Multiplication Kernel

```c
__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;
}```
How about performance on 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
  - $4 \times 1000 = 4000$ 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

Sharing Global Memory Accesses

- Note the number of global memory accesses to row 0
- Can we coordinate timing between threads $P_{0,0}$ and $P_{0,1}$
- In general, will see $N$ copies for a $N\times N$ thread block

Row = 0
Row = 1
Sharing Global Accesses

- Good – when threads have similar access timing
- Bad – when threads have very different timing

Shared Memory Blocking Basic Idea

- Collaboratively load values
- Access/execute in parallel
Outline of Technique

- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile
- Synchronization to enforce timing/ordering

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
Work for Block (0,0) in a TILE_WIDTH = 2 Configuration

Col = 0 * 2 + threadIdx.x
Row = 0 * 2 + threadIdx.y

Row = 0
Row = 1

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
• Same basic algorithm, i.e., each thread computes one element of output matrix
  – Add optimized loading from main memory
Loading a Tile

- Basic approach for each thread block
  - Parallel load of tile elements
  - Synchronize
  - Compute partial product
  - Synchronize
  - Iterate
- All threads in a block participate
  - Each thread loads one Md element and one Nd element in based tiled code
- Coalesce accesses to memory (later)
Work for Block (0,0)

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012

33
Work for Block (0,0)

N_{0,0} \ N_{0,1} \ N_{0,2} \ N_{0,3}
N_{1,0} \ N_{1,1} \ N_{1,2} \ N_{1,3}
N_{2,0} \ N_{2,1} \ N_{2,2} \ N_{2,3}
N_{3,0} \ N_{3,1} \ N_{3,2} \ N_{3,3}

SM

M_{0,0} \ M_{0,1} \ M_{0,2} \ M_{0,3}
M_{1,0} \ M_{1,1} \ M_{1,2} \ M_{1,3}
M_{2,0} \ M_{2,1} \ M_{2,2} \ M_{2,3}
M_{3,0} \ M_{3,1} \ M_{3,2} \ M_{3,3}

P_{0,0} \ P_{0,1} \ P_{0,2} \ P_{0,3}
P_{1,0} \ P_{1,1} \ P_{1,2} \ P_{1,3}
P_{2,0} \ P_{2,1} \ P_{2,2} \ P_{2,3}
P_{3,0} \ P_{3,1} \ P_{3,2} \ P_{3,3}

© David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Barrier Synchronization

- An API function call 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

Bars and Thread Behaviors

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

... Thread N-3
Thread N-2
Thread N-1

Time →
Loading an Input Tile

Accessing tile 0 in 2D indexing:
M[Row][tx]
N[ty][Col]

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:

\[
\begin{align*}
M[\text{Row}] &\left[ m\cdot\text{TILE \_ WIDTH} + tx \right] \\
M[\text{Row} \cdot \text{Width} + m\cdot\text{TILE \_ WIDTH} + tx] \\
N[m\cdot\text{TILE \_ WIDTH} + ty][\text{Col}] \\
N[(m\cdot\text{TILE \_ WIDTH} + ty) \cdot \text{Width} + \text{Col}]
\end{align*}
\]

Tiled Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {
  1. __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  2. __shared__ float ds_N[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;
  // Loop over the Md and Nd tiles required to compute the Pd element
  7. float Pvalue = 0;
  8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    9.  ds_M[ty][tx] = d_M[Row*Width + m*TILE_WIDTH+tx];
   10. ds_N[ty][tx] = d_N[Col+(m*TILE_WIDTH+ty)*Width];
   11.  __syncthreads();
   12.  for (int k = 0; k < TILE_WIDTH; ++k)
   13.    Pvalue += ds_M[ty][k] * ds_N[k][tx];
   14.  __syncthreads();
   15. }
   16.  d_P[Row*Width+Col] = Pvalue;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2011
Compare with the Base Kernel

```c
__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;
}
```

First-order Size Considerations

- Each **thread block** should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads

- For 16, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.

- For 32, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations
Shared Memory and Threading

• Each SM in Fermi has 16KB or 48KB shared memory*
  – SM 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 86.4GB/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

*Configurable vs L1, total 64KB

Resource Constraints

• Number of available registers also limit the number of thread blocks that can concurrently execute
  – Reduction is in size of a thread block
  – Impact on utilization can be significant
• Auto-tune based on querying of device properties
  – E.g., fix tile size based on available shared memory size
  – Need to change the preceding tiled MM code
Device Query

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

- Capability of devices
  
  ```c
  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
  - ...

Summary - Typical Structure of a CUDA Program

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

- Function prototypes
  
  ```c
  __global__ void kernelOne(…)
  ```

- Main()
  
  ```c
  allocate memory space on the device – cudaMalloc( &d_GlblVarPtr, bytes )
  ```

- Other functions
  
  ```c
  repeat
  ```
ANY MORE QUESTIONS?
READ CHAPTER 5!