Reading Assignment

- Kirk and Hwu, “Programming Massively Parallel Processors: A Hands on Approach,”, Chapter 6
- CUDA Programming Guide
Objective

• To understand the implications of programming model constructs on demand for execution resources
• To be able to reason about performance consequences of programming model parameters
  – Thread blocks, warps, memory behaviors, etc.
  – Need deeper understanding of architecture to be really valuable (later)
• To understand DRAM bandwidth
  – Cause of the DRAM bandwidth problem
  – Programming techniques that address the problem: memory coalescing, corner turning,

Formation of Warps

• How do you form warps out of multidimensional arrays of threads?
  – Linearize thread IDs
Formation of Warps

3D Thread Block

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Thread (0,0,0)

Thread (0,0,1)

Thread (0,0,2)

Thread (0,0,3)

Thread (0,1,0)

Thread (0,1,1)

Thread (0,1,2)

Thread (0,1,3)

Block (1,1)

2D Thread Block

T0,0,0 T0,0,1 T0,0,2 T0,0,3 T0,1,0 T0,1,1 T0,1,2 T0,1,3 T1,0,0 T1,0,1 T1,0,2 T1,0,3 T1,1,0 T1,1,1 T1,1,2 T1,1,3

linear order

Execution of Warps

- Each warp executed as SIMD bundle
- How do we handle divergent control flow among threads in a warp?
  - Execution semantics
  - How is it implemented? (later)

© David Kirk / NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al, 2007-2012
Reduction: Approach 1

1. `__shared__ float partialsum[];`

2. `unsigned int t = threadIdx.x;`

3. `for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)`

4. `while (true)`

5. `__syncthreads();`

6. `if (t % (2 * stride) == 0)`

7. `partialsum[t] += partialsum[t + stride];`

8. `}`

Reduction: Approach 2

1. `__shared__ float partialsum[];`

2. `unsigned int t = threadIdx.x;`

3. `for (unsigned int stride = blockDim.x; stride > 1; stride /= 2)`

4. `while (true)`

5. `__syncthreads();`

6. `if (t < stride)`

7. `partialsum[t] += partialsum[t + stride];`

8. `}`

- Difference is in **which** threads diverge!
- For a thread block of 512 threads
  - Threads 0-255 take the branch, 256-511 do not
- For a warp size of 32, all threads in a warp have identical branch conditions $\rightarrow$ no divergence!
- When #active threads < warp-size, $\rightarrow$ old problem
Global Memory (DRAM) Bandwidth

Ideal

Reality

©Wen-mei W. Hwu and David Kirk/NVIDIA, ECE408/CS483/ECE498AL, University of Illinois, 2007-2012

DRAM Bank Organization

- Each core array has about 1M bits
- Each bit is stored in a tiny capacitor, made of one transistor

©Wen-mei W. Hwu and David Kirk/NVIDIA, ECE408/CS483/ECE498AL, University of Illinois, 2007-2012
A very small (8x2 bit) DRAM Bank

0 1 1

decode

Sense amps

Mux

DRAM core arrays are slow.
- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - ... likely to be worse in the future

decode

About 1000 cells connected to each vertical line

A very small capacitance that stores a data bit

To sense amps
DRAM Bursting.

- For DDR{2,3} SDRAM cores clocked at 1/N speed of the interface:
  - Load \((N \times \text{interface width})\) of DRAM bits from the same row at once to an internal buffer, then transfer in \(N\) steps at interface speed
  - DDR2/GDDR3: buffer width = 4X interface width
DRAM Bursting

"You can buy bandwidth but you can’t bribe God”
-- Unknown

DRAM Bursting for the 8x2 Bank

Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

©Wen-mei Hwu and David Kirk/NVIDIA, ECE408/CS485, University of Illinois, 2007-2012
Multiple DRAM Banks

Bank 0

Sense amps

Mux

Bank 1

Sense amps

Mux

DRAM Bursting for the 8x2 Bank

Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time
First-order Look at the GPU off-chip memory subsystem

- nVidia GTX280 GPU:
  - Peak global memory bandwidth = 141.7GB/s

- Global memory (GDDR3) interface @ 1.1GHz
  - (Core speed @ 276Mhz)
  - For a typical 64-bit interface, we can sustain only about 17.6 GB/s (Recall DDR - 2 transfers per clock)
  - We need a lot more bandwidth (141.7 GB/s) – thus 8 memory channels

Multiple Memory Channels

- Divide the memory address space into N parts
  - N is number of memory channels
  - Assign each portion to a channel
Memory Controller Organization of a Many-Core Processor

- GTX280: 30 Stream Multiprocessors (SM) connected to 8-channel DRAM controllers through interconnect
  - DRAM controllers are interleaved
  - Within DRAM controllers (channels), DRAM banks are interleaved for incoming memory requests

Lessons

- Organize data accesses to maximize burst mode bandwidth
  - Access consecutive locations
  - Algorithmic strategies + data layout
- Thread blocks issue warp-size load/store instructions
  - 32 addresses in Fermi
  - Coalesce these accesses to create smaller number of memory transactions \(\rightarrow\) maximize memory bandwidth
  - More later as we discuss microarchitecture
Memory Coalescing

- Memory references are coalesced into sequence of memory transactions
  - Accesses to a segment are coalesced, e.g., 128 byte segments)

Implications of Memory Coalescing

- Reduce the request rate to L1 and DRAM
- Distinct from CPU optimizations – why?
- Need to be able to re-map entries from each access back to threads
Placing a 2D C array into linear memory space

![Diagram of 2D array linearization](image)

**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;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu,
ECE488/CS483/ECE498al, University of Illinois, 2007-2012
Two Access Patterns

\[ d_M[\text{Row} \times \text{Width} + k] \quad d_N[k \times \text{Width} + \text{Col}] \]

\( k \) is loop counter in the inner product loop of the kernel code

N accesses are coalesced.

Across successive threads in a warp

Access direction in kernel code (one thread)

\( d_N[k \times \text{Width} + \text{Col}] \)
M accesses are not coalesced.

Access across successive threads in a warp

\[ d_M[\text{Row*Width+k}] \]

Load iteration 0

Load iteration 1

Using Shared Memory

Original Access Pattern

Tiled Access Pattern

Copy into scratchpad memory

Perform multiplication with scratchpad values
Shared Memory Accesses

- Shared memory is banked
  - No coalescing
- Data access patterns should be structured to avoid bank conflicts
- Low order interleaved mapping?

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, 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 d_P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute the d_P element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        Mds[tx][ty] = d_M[Row*Width + m*TILE_WIDTH+tx];
        Nds[tx][ty] = d_N[(m*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[tx][k] * Nds[k][ty];
    }
    __syncthreads();
    d_P[Row*Width+Col] = Pvalue;
}
```

- Accesses from shared memory, hence coalescing is not necessary
- Consider bank conflicts
Coalescing Behavior

- Consider instruction bandwidth vs. memory bandwidth
- Control amount of work per thread
Thread Granularity Tradeoffs

- Preserving instruction bandwidth (memory bandwidth)
  - Increase thread granularity
  - Merge adjacent tiles: sharing tile data

Thread Granularity Tradeoffs (2)

- Impact on parallelism
  - #TBs, #registers/thread
  - Need to explore impact → autotuning
ANY MORE QUESTIONS?
READ CHAPTER 6!