ECE 8823A

GPU Architectures

Module 5: Execution and Resources - I

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,

Closer Look: 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 (0, 1)
Block (1, 0)
Block (1, 1)

2D Thread Block

3D Thread Block

Mapping Thread Blocks to Warps

An Example with a warp size of 16 threads

- Follow row major order through the Z-dimension
- Linearize and then split into warps
- Understanding becomes important when optimizing global memory accesses
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)
  - How can we optimize against it?

Impact of Control Divergence

- Occurs within a warp
- Branches lead serialization branch dependent code
  - Performance issue: low warp utilization

```c
if(...) {
    ...
} else {
    ...
}
```
Causes

- Traditional nested branches
- Loops
  - Variable number of iterations/thread
  - Loop condition based on thread ID?
- Switching on thread ID
  \[
  \text{if(threadID}\_x > 5) \}\{}
\]

Control Divergence Mitigation: Algorithmic Approach

Loosely synchronized threads

Flexibility of MIMD control flow + Benefits of SIMD execution

Can algorithmic techniques maximize utilizations achieved by a warp?
Reduction

• A commonly used strategy for processing large input data sets
  – There is no required order of processing elements in a data set (associative and commutative)
  – Partition the data set into smaller chunks
  – Have each thread to process a chunk
  – Use a reduction tree to summarize the results from each chunk into the final answer
• We will focus on the reduction tree step for now.
• Google and Hadoop MapReduce frameworks are examples of this pattern

A parallel reduction tree algorithm performs N-1 Operations in log(N) steps
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. |
5. `__syncthreads();`
6. If(t%(2*stride) == 0)
7. `partialsum[t] += partialsum[t+stride];`

A Better Strategy

- **Principle**: Shift the index usage to ensure high thread utilization in warp
  - Remap thread indices

- Keep the active threads consecutive
An Example of 16 threads

Thread 0  Thread 1  Thread 2  Thread 14  Thread 15

No Divergence

0  1  2  3  ...  13  14  15  16  17  18  19

0+16  15+31

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. {
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 → no divergence!
• When #active threads <warp-size, → old problem
Global Memory Bandwidth

• How can we map thread access patterns to global memory addresses to maximize bandwidth utilization?
• Need to understand the organization of DRAMs!
  – Hierarchy of latencies

Basic Organization

Example: 32x32 = 1024 bit array
Technology Trends

Past two decades,
- Data rate increase $\sim 1000\times$
- RAS/CAS latency decrease $= 56\%$

How? $\rightarrow$ increasing burst length
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.
DRAM Bursting for the 8x2 Bank

Address bits to decoder

Core Array access delay

2 bits 2 bits to pin to pin

time

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 V100 Volta GPU:
  - Peak global memory bandwidth = 900 GB/s

- Global memory (HBM2) interface @ 4096 bits

- Prior generation GPUs (e.g., Kepler) 384 bit wide
  GDDR5 @ 224GBytes/sec
Multiple Memory Channels

- Divide the memory address space into \( N \) parts
  - \( N \) is number of memory channels
  - Assign each portion to a channel

“*You can buy bandwidth but you can’t bribe God*”

-- Unknown

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 for a warp size of 32
  - **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)
- Ability and extent of coalescing depends on compute capability

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

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;
}
```
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)

Load iteration 0
\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]

Load iteration 1
\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]

...
M accesses are not coalesced.

Access direction in Kernel code (in a thread)

Access across successive threads in a warp

Load iteration 1

T0  T1  T2  T3

T0  Load iteration 0  T1  T2  T3

M

M0,0  M0,1  M0,2  M0,3  M1,0  M1,1  M1,2  M1,3  M2,0  M2,1  M2,2  M2,3  M3,0  M3,1  M3,2  M3,3

...  d_M[Row*Width+k]  ...

Using Shared Memory

Original Access Pattern

d_M  d_N

Tiled Access Pattern

d_M  d_N

Copy into scratchpad memory

Perform multiplication with scratchpad values

©Wen-mei W. Hwu and David Kirk/NVIDIA, ECE408/CS483/ECE496AL, University of Illinois, 2007-2012
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) {
        // Collaborative loading of d_M and d_N tiles into shared memory
        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

Thread Granularity

- 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!