CS/EE 217
GPU Architecture and Parallel Programming

Lecture 6: DRAM Bandwidth
Objective

• To understand DRAM bandwidth
  – Cause of the DRAM bandwidth problem
  – Programming techniques that address the problem: memory coalescing, corner turning,
Global Memory (DRAM) Bandwidth

Ideal

Reality
DRAM Bank Organization

- Each core array has about 1M bits
- Each bit is stored in a tiny capacitor, made of one transistor
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 = $\frac{1}{2}$ interface speed
  - DDR2/GDDR3: Core speed = $\frac{1}{4}$ interface speed
  - DDR3/GDDR4: Core speed = $\frac{1}{8}$ interface speed
  - … likely to be worse in the future

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

0 1 0

decode

Sense amps

Mux
DRAM Bursting

0 1 1

decode

Sense amps and buffer

Mux
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.
Multiple DRAM Banks

Bank 0

Bank 1
DRAM Bursting for the 8x2 Bank

Address bits to decoder

Core Array access delay

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 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
Placing a 2D C array into linear memory space
__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

Thread 1
Thread 2

(a)
(b)

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

k is loop counter in the inner product loop of the kernel code.
N accesses are coalesced.
M accesses are not coalesced.

Access direction in Kernel code

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

Load iteration 1

Load iteration 0
__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;
}
Figure 6.10: Using shared memory to enable coalescing
ANY MORE QUESTIONS?
READ CHAPTER 6