CUDA: Matrix Multiplication

Mark Greenstreet

CpSc 421 – Mar. 18, 2016

- Remarks about the Homework
- The Brute Force Approach

Remarks about the homework

- Yes, you are allowed to base your solutions on code I have presented in class.
  - Of course, you must add a comment saying that's what you did.
  - Include a summary of the changes you made and why.
- For each problem, focus on the aspect it is trying to measure.
  - E.g. Q1 is for measuring FLOPS – anything else is irrelevant.
  - E.g. Q2 is for measuring memory bandwidth
    - Observe that there is only one floating point operation, a fused multiply-add, per memory read.
    - With $>$ 100 SPs, is should be "easy" to make the memory reads the main bottleneck.
    - You can delegate the final sums (over blocks, and even over threads) to the CPU.
    - For timing, just measure the GPU part – that's where the memory reads are happening.
  - E.g. Q3 measures calls to the random number generator library.

Brute-force matrix multiplication

- Brute-force, data-parallel: one thread per element of the result.
- matrixMult: compute c = a*b
- For simplicity, assume all matrices are $n \times n$.

```
__global__ matrixMult(float *a, float *b, float *c, int n) {
   float *a_row = a + (blockDim.y*blockIdx.y + threadIdx.y)*n;
   float *b_col = b + (blockDim.x*blockIdx.x + threadIdx.x);
   int myY = blockDim.y*blockIdx.y + threadIdx.y;
   float sum = 0.0;
   for(int k = 0; k < n; i++) {
   sum += (*a_row) * (*b_col);
   a_row = a_row+1;
   b_col = a_col+n;
   }
   c[ (blockDim.y*blockIdx.y + threadIdx.y)*n +
      (blockDim.x*blockIdx.x + threadIdx.x) ] = sum;
}
```

Brute-force performance

- Not very good.
- Each loop iteration performs
  - Two global memory reads.
  - One fused floating-point add.
  - Four or five integer operations.
- Global memory is slow
  - Long access times.
  - Bandwidth shared by all the SPs.
- This implementation has a low **CGMA**
  - CGMA = Compute to Global Memory Access ratio.

Tiling the computation

- Divide eacn matrix into $m \times m$ tiles.
  - For simplicity, we'll assume that n is a multiple of m.
- Each block computes a tile of the product matrix.
  - Computing a $m \times m$ tile involves computing n/m products of $m \times m$ tiles and summing up the results.

A Tiled Kernel

Based on Fig. 6.11 in the text.

```
#define TILE_WIDTH 16 __global__ matrixMult(float *a, float *b, f
   __shared__ float sh_a[TILE_WIDTH][TILE_WIDTH];
   __shared__ float sh_b[TILE_WIDTH][TILE_WIDTH];
   int n_tiles = n / TILE_WIDTH; float sum = 0.0;
   for(int kk = 0; kk < n_tiles; kk++) { % each tile produce
      % load tiles
      __syncthreads();
      % compute product
      }
      c[ (blockDim.y*blockIdx.y + threadIdx.y)*n +
         (blockDim.x*blockIdx.x + threadIdx.x) ] = sum;
   }
```