CUDA: Matrix Multiplication

Mark Greenstreet

CpSc 418 – Mar. 23, 2016

- Makefiles, etc.
- The Brute Force Approach

Makefiles

```
# definitions
NVCC    = nvcc
CFLAGS  = -O3
LDFLAGS =
OBJ     = time_it.o

default:  hw3

examples:  examples.o $(OBJ)
        $(NVCC) $(CFLAGS) examples.o $(OBJ) $(LDFLAGS) -o examples

hw3:  hw3.o $(OBJ)
        $(NVCC) $(CFLAGS) hw3.o $(OBJ) $(LDFLAGS) -o hw3

.SUFFIXES: .c .cu .o
.c.o:
        $(NVCC) -c $(CFLAGS) $<
.cu.o:
        $(NVCC) -c $(CFLAGS) $<
```

Extend the homework deadline?

What do you think?

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);
   float sum = 0.0;
   for(int k = 0; k < n; k++) {
      sum += a_row[k] * b_col[n*k];
   }
   c[ (blockDim.y*blockIdx.y + threadIdx.y)*n +
      (blockDim.x*blockIdx.x + threadIdx.x) ] = sum;
}
```

Launching the kernel:

```
   int nblks = n/blk_size;
   dim3 blks(nblks, nblks, 1);
   dim3 thrds(blk_size, blk_size, 1);
   matrixMult<<<blks,thrds>>>(a, b, c, n);
```

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 each 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 (step 1)

```
#define TILE_WIDTH 16
__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);
   float sum = 0.0;
   for(int k1 = 0; k1 < gridDim.x; k1++) { % each tile product
      for(int k2 = 0; k2 < blockDim.x; k2++) { % within each tile
         k = k1*blockDim.x + k2;
         sum += a_row[k] * b_col[n*k]);
      }
   }
   c[ (blockDim.y*blockIdx.y + threadIdx.y)*n +
      (blockDim.x*blockIdx.x + threadIdx.x) ] = sum;
}
```

Launching the kernel:

```
   int nblks = n/TILE_WIDTH;
   dim3 blks(nblks, nblks, 1);
   dim3 thrds(TILE_WIDTH, TILE_WIDTH, 1);
   matrixMult<<<blks,thrds>>>(a, b, c, n);
```

A Tiled Kernel (step 2)

```
__global__ matrixMult(float *a, float *b, float *c, int n) {
   __shared__ a_tile[TILE_WIDTH][TILE_WIDTH];
   __shared__ b_tile[TILE_WIDTH][TILE_WIDTH+1];
   int br = blockIdx.y,     bc = blockIdx.x;
   int tr = threadIdx.y,    tc = threadIdx.x;
   float *a_row = a + (blockDim.y*br + tr)*n;
   float *b_col = b + (blockDim.x*bc + tc);
   float sum = 0.0;
   for(int k1 = 0; k1 < gridDim.x; k1++) { % each tile product
      a_tile[tr][tc] = a_row[TILE_WIDTH*k1 + tc];
      b_tile[tr][tc] = b_col[n*(TILE_WIDTH*k1 + tr)];
      __syncthreads();
      for(int k2 = 0; k2 < blockDim.x; k2++) { % within each tile
         sum += a_tile[tc][k2] * b_tile[k2][tc];
      }
      __syncthreads();
   }
   c[(blockDim.y*br + tr)*n + (blockDim.x*bc + tc) ] = sum;
}
```

- Launching the kernel: same as on slide 7.
- See also, Kirk & Hwu, Fig. 6.11 (p. 110).

Coalesced Memory Addresses

- Note: I've written `r` for "row" and "c" for column instead of x and y when defining `br`, `bc`, `tr`, and `tc`.
- The memory accesses are coalesced!
  - Linearizing the thread indices:

    ```
    linearIndex = blockDim.x*threadIdx.y + threadIdx.x
    ```
  - Reading from `a_row`
    - `a_tile[tr][tc] = a_row[TILE_WIDTH*k1 + tc];`
    - Consecutive threads have consecutive indices for tc.
    - The references are coalesced.
    - Note: one warp has threads for two rows: not *perfectly* coalesced.
  - Reading from `b_col`
    - `b_tile[tr][tc] = b_col[n*(TILE_WIDTH*k1 + tr)];`
    - Consecutive threads have consecutive indices for `b_col` pointers.
    - The references are coalesced (same remark about not quite perfect).
  - Writing to `b`
    - Not a big deal. Why?
    - Even so, the writes are coalesced.

Could we do better?

Sure.

- Prefetch: hide memory latency.
- Double-buffer the tiles: avoid a syncthreads.
- Use larger blocks: perfect coalescing.
- Do we have enough shared memory?
  - Current version stores 256 float in `a_tile` and 256 in `b_tile` for a total of 2K bytes.
  - To keep the SM fully occupied, we need 6 blocks per SM. That's 12K bytes.
  - With optimizations:
    - ★ Double buffering uses 24K bytes of shared memory per SM.
    - ★ 32×32 blocks use 48K bytes of shared memory per SM.
    - ★ Doing both uses 96K bytes of shared memory per SM.
- We might be able to do both if we made each thread compute **two** elements of the result.
  - Need to write the code and make timing measurements before trying fancy optimizations.

Tiling is good for more than just matrix multiplication

- Other numerical applications:
  - LU-decomposition and other factoring algorithms.
  - Matrix transpose.
  - Finite-element methods.
  - Many, many more.
- A non-numerical example: `revsort`

```
% To sort $N^2$ values, arrange them as a $N \times N$ array.
repeat $\log N$ times {
   sort even numbered rows left-to-right.
   sort odd numbered rows right to left.
   sort columns top-to-bottom.
}
```

  - We can get coalesced accesses for the rows, but not the columns.
  - Cooperative loading can help here – e.g. use a transpose.

Summary

- Brute-force matrix multiplication is limited by global memory bandwidth.
- Using tiles addresses this bottleneck:
  - Load tile into shared memory and use them many times.
  - Each tile element is used by multiple threads.
  - The threads cooperate to load the tiles.
  - This approach also provides memory coalescing.
- Other optimizations: prefetching, double-buffering, loop-unrolling.
  - First, identify the critical bottleneck.
  - Then, optimize.
- These ideas apply to many parallel programming problems:
  - When possible, divide the problem into blocks to keep the data local.
  - Examples include matrix and mesh algorithms.
  - The same approach can be applied to non-numerical problems as well.

Preview

The rest of the term:

- Parallel sorting
  - Sorting networks and the 0-1 principle.
  - Application to parallel sorting: bitonic sort.
- Other stuff
  - map-reduce and hadoop.
  - That's probably all the time we'll have.