CUDA: Matrix Multiplication

Mark Greenstreet

CpSc 418 – Mar. 24, 2017

- A Brute Force Implementation
- Tiling

`mmult1`: brute-force matrix multiplication

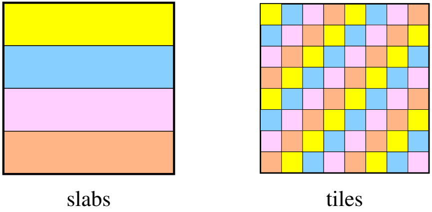
The kernel:

```
% one thread per element of the result.
% matrixMult: compute c = a*b
% For simplicity, assume all matrices are n × n.
__global__ mmult1_kernel(float *a, float *b, float *c, uint n) {
   uint i = blockDim.y*blockIdx.y + threadIdx.y;
   uint j = blockDim.x*blockIdx.x + threadIdx.x;
   if((i < n) && (j < n)) {
      float *a_row = a + n*i;
      float *b_col = b + j;
      float sum = 0.0;
      for(int k = 0; k < n; k++) {
         sum += a_row[k] * b_col[n*k];
      }
      c[i*n + j] = sum;
   }
}
```

Brute-force performance

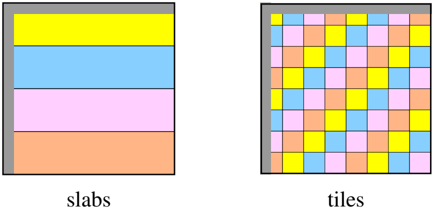
- Not very good – each loop iteration performs
  - Two global memory reads.
  - One fused floating-point add.
  - Four or five integer operations.
- Global memory is slow
  - Long access times.
  - Bandwidth shared by all the SPs.
- This implementation has a low **CGMA**
  - CGMA = Compute to Global Memory Access ratio $\approx$ 1/2.
- Performance should be:
  - asymptotics: $\mathcal{O}(N^3)$
  - wall-clock: $\sim \alpha N^3$ with α determined mainly by global memory bandwidth.
  - measured: $T(1024) \approx 0.0986\mathrm{s}$; $T(2048) \approx 0.797\mathrm{s}$; $T(3072) \approx 2.7\mathrm{s}$; $T(4096) \approx 6.3\mathrm{s}$.
    $N^3/T(N) \approx 11/\mathrm{ns}$ – i.e. about 20×10^9 multiply-adds per second.
    Well below GPU peak floating point capacity. Demonstrates global memory bandwith bottleneck (with a little help from the on-chip caches).

Tiles vs. Slabs



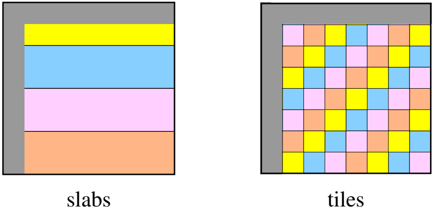
- Matrix multiplication: each processor (color) has tiles at (i,j) and (j,i).
  - Can compute all products for the main diagonal, and stripes at spacings of P.
  - Use a reduce to combine results to get the main diagonal and the stripes.
  - Rotate B one block to the left, and compute the next set of strips.
  - After P rounds, the computation is done.
  - Same amount of work (and communication) as the improved slab method from Wednesday.
- Other algorithms such as LU-Decomposition
  - Rows and columns are eliminated from the left and the top.
  - Tiles provide better load balancing.

Tiles vs. Slabs



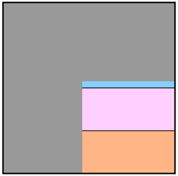
- Matrix multiplication: each processor (color) has tiles at (i,j) and (j,i).
  - Can compute all products for the main diagonal, and stripes at spacings of P.
  - Use a reduce to combine results to get the main diagonal and the stripes.
  - Rotate B one block to the left, and compute the next set of strips.
  - After P rounds, the computation is done.
  - Same amount of work (and communication) as the improved slab method from Wednesday.
- Other algorithms such as LU-Decomposition
  - Rows and columns are eliminated from the left and the top.
  - Tiles provide better load balancing.

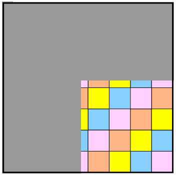
Tiles vs. Slabs



- Matrix multiplication: each processor (color) has tiles at (i,j) and (j,i).
  - Can compute all products for the main diagonal, and stripes at spacings of P.
  - Use a reduce to combine results to get the main diagonal and the stripes.
  - Rotate B one block to the left, and compute the next set of strips.
  - After P rounds, the computation is done.
  - Same amount of work (and communication) as the improved slab method from Wednesday.
- Other algorithms such as LU-Decomposition
  - Rows and columns are eliminated from the left and the top.
  - Tiles provide better load balancing.

Tiles vs. Slabs





- Matrix multiplication: each processor (color) has tiles at (i,j) and (j,i).
  - Can compute all products for the main diagonal, and stripes at spacings of *P*.
  - Use a reduce to combine results to get the main diagonal and the stripes.
  - Rotate *B* one block to the left, and compute the next set of strips.
  - After *P* rounds, the computation is done.
  - Same amount of work (and communication) as the improved slab method from Wednesday.
- Other algorithms such as LU-Decomposition
  - Rows and columns are eliminated from the left and the top.
  - Tiles provide better load balancing.

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__ mmult2(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;
}
```

Performance of `mmult2`

- $T(1024) = 0.027\text{s}$; $T(2048) = 0.214\text{s}$; $T(3072) = 0.742\text{s}$; $T(4096) = 1.73\text{s}$.
- Still cubic in N, of course.
  - $N^3/T(N) \approx 40/\text{ns}$ – about 40 billion multiply-adds per second.
  - About four times faster than `mmult1`.

Performance issues for `mmult2`

The "checklist"

- Are global memory accesses coalesced?
- What is the CGMA?
- Do we have shared memory access conflicts?
- What is the warp-scheduler occupancy?
  - How many registers per thread?
  - How many threads per block?
  - How much shared memory per block?
- How much "other stuff" does each thread perform for each floating point operation?

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 × 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

March 27: Using Parallel Libraries	
March 29: Introduction to Model Checking	
Reading:	Protocol Verification as a Hardware Design Aid
March 31: The PReach Model Checker	
Reading:	Industrial Strength … Model Checking
April 3: Distributed Termination Detection	
April 5: Party: 50^{th} Anniversary of Amdahl's Law	