Lecture 8

Further improvements to Matrix Multiply
Announcements

• Mac Mini lab (APM 2402)
  - Friday at 4pm to 6pm
  - Next week: Tues and Fri 4pm to 6pm

• Project proposals due on November 9
Projects!

- Stencil method in 3 dimensions
- Multigrid
- Communication avoiding matrix multiplication (MPI)
- Algorithm based fault tolerance (MPI)
- 3D Fast Fourier Transform (MPI or CUDA)
- Particle simulation (MPI)
- Groups of 3 will do a more ambitious project
  - MPI projects can add communication overlap
  - MPI + CUDA
- Propose your own
- Make your choice by 11/9

www-cse.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
Today’s lecture

• Bank conflicts (previous lecture)
• Further improvements to Matrix Multiplication
• Thread divergence
• Scheduling
Fermi platforms in the class

CSEClass 01, 02: GeForce GTX 580 [2.0, GF100]
  15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
  1.25 GB device memory
CSEClass 03-07: GeForce GTX 460 [2.1, GF104]
  7 Vector units @ 48 cores (384 total cores), 8 SFUs
  1.0 GB device memory
Dirac: Tesla C2050 [2.0, GF100]
  1 device per node
  14 Vector units @ 32 cores (448 total cores), 4 SFUs
  3 GB device memory + ECC (2.625GB usable)
  SP MAD: 1030.4 Gflops, DP FMA: 515.2

www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2
Warp Scheduling (Fermi)

- Threads assigned to an SM in units of a thread block, multiple blocks
- Each block is divided into *warps* of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a Warp execute the same instruction, branches serialize execution
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SM
- How does scheduling work?
Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?
Shared memory banks

- A load or store of $n$ addresses spanning $n$ distinct memory banks can be serviced simultaneously, effective bandwidth is $\times n$ single bank bandwidth
- Multiple addresses map to same memory bank
  - Accesses are serialized
  - Hardware splits request into as many separate conflict-free requests as necessary
    Exception: if all access the same address: broadcast
- Devices of compute capability 2.x have the additional ability to multicast shared memory accesses
- See *CUDA C Best Practices Guide*
Shared memory bank access

- Load/store of \( n \) addresses spanning \( n \) distinct memory banks can be serviced simultaneously, effective BW = \( \times n \) a single bank
- Each bank can service 1 address / cycle (broadcast, too)
- Access to shared memory is fast unless…
  - 2 or more instructions in a 1/2 warp access different banks: we have a conflict
  - Exception: if all access the same bank: broadcast

\[
\text{int idx} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}; \\
\text{a[idx]} = \text{a[idx]} + 1.0f;
\]
Identifying bank conflicts

- Traditional wisdom for exploiting cache locality can result in bank conflicts.
- What if a thread loads 2 consecutive array elements?
  ```
  int tid = threadIdx.x;
  shared[2*tid] = global[2*tid];
  shared[2*tid+1] = global[2*tid+1];
  ```
- To avoid conflicts
  ```
  shared[tid] = global[tid];
  shared[tid + blockDim.x] = global[tid + blockDim.x];
  ```
- Consider
  ```
  __shared__ float shared[256];
  float foo = shared[base + s * threadIdx.x];
  ```
- If \( s \) has no common factors with the number of banks (16), then there are no conflicts (\( s \) is odd).
Shared memory design

- Successive 32-bit words assigned to successive banks
- For devices of compute capability 2.x [Fermi]
  - Number of banks = 32
  - Bandwidth is 32 bits per bank per 2 clock cycles
  - Shared memory request for a warp is not split
  - Increased susceptibility to conflicts
  - But no conflicts if access to bytes in same 32 bit word

- For devices of compute capability 1.x [Lilliput]
  - Number of banks = 16
  - Bandwidth is 32 bits per bank per clock cycle
  - Shared memory request for a warp is split in two
  - No conflict occurs if only one memory location per bank is accessed by a half warp of threads
Coalesced access and no bank conflicts

\[ I = blockIdx.y \times by + ty; \]
\[ J = blockIdx.x \times bx + tx; \]

\[
\text{__shared__ float } a[\text{BLK}][\text{BLK}], \quad b[\text{BLK}][\text{BLK}];
\]
\[
\text{if } ((I < N) \land \land (J < N))\{ \text{ float c = 0.0f; for (k=0; k < gy; k++)\{ a[ty][tx] = A[I \times N + k \times by + tx]; b[ty][tx] = B[J + N \times (k \times bx + ty)]; \text{__syncthreads(); for (kk=0; kk < bx; kk++) c += a[ty][kk] \times b[kk][tx]; __syncthreads(); \}}}
\]
\[ C[I \times N + J] = c; \]
Today’s lecture

• Bank conflicts (previous lecture)
• Further improvements to Matrix Multiplication
• Thread divergence
• Scheduling
How to improve matrix multiply

• Volkov and Demmel, SC08
• Hide arithmetic latency using fewer threads
• Hide memory latency using fewer threads
• Improving performance using fewer threads
  • We can reduce number of threads through lower occupancy
  • By making better use of registers we can trade locality against parallelism
Latency

• Instructions wait on dependencies
  \( x = a + b; \)  // ~20 for floating point, 500+ for memory
  \( y = a + c; \)  // independent (stall)
  \( z = x + d; \)  // dependent, wait

• How many warps are needed to hide latency
if minimum latency is 4 cycles / instruction?
  \[ \# \text{Parallelism (threads)} = \text{latency} \times \text{throughput} \]
  \[ T = \lambda \times p \]
  480 mul-adds/cycle; 32 memory ops /cycle [1.3 device]

• Required parallelism depends on op; for single precision
  ♦ GT200 (C1060, Lilliput): 24 CP * 8 cores / SM = 192 ops/cycle
  ♦ GF100 (GTX-580, Cseclass01/02): 18 CP * 32 = 576
  ♦ GF104 (GTX 460, Cseclass03-07): 18 CP * 48 = 864
Thread vs instruction level parallelism

- We are told to maximize the number of threads
- But we can also use instruction level parallelism to boost performance at a lower occupancy
  - See http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- On GT200, 100% peak with 25% occupancy
  192 ops / cycle = 8 warps / 32 max possible warps
- On the GF104, we need ILP to go beyond 66% of peak
  - 48 cores/SM, half warp (16 cores) issues at a time
  - But we have only 2 schedulers
  - We must issue 2 independent instructions per warp in the same cycle

576 threads needed for 100% utilization

 pragma unroll UNROLL
 for( i = 0; i < N_Iter; i++ ){
   a = a * b + c;
 }

320 threads needed for 100% utilization

 pragma unroll UNROLL
 for( i = 0; i < N_Iter; i++ ){
   a = a * b+c;
   d = d * b + c;
 }

10/24/12  ©2012 Scott B. Baden /CSE 260/ Fall 2012
Hiding memory latency

- **Parallelism = latency \times throughput**

  Arithmetic: \(576\ \text{ops/SM} = 18\times 32/\text{SM/CP}\)  
  Memory: \(150\text{KB} = \sim 500\text{CP} \ (1100\ \text{nsec}) \times 150\ \text{GB/sec}\)

- **How can we keep 150KB in flight?**
  - Multiple threads: \(~35,000\ \text{threads} @ 4\text{B/thread}\)
  - ILP (increase fetches per thread)
  - Larger fetches (64 or 128 bit/thread)
  - Higher occupancy

Copy 1 float /thread, need 100% occupancy
\[
\text{int}\ \text{indx} = \text{threadIdx.x} + \text{block} \times \text{blockDim.x}; \\
\text{float a0} = \text{src}[\text{indx}]; \\
\text{dest}[\text{indx}] = \text{a0};
\]

Copy 2 floats /thread, need 50% occ
\[
\text{float a0} = \text{src}[\text{indx}]; \\
\text{float a1} = \text{src}[\text{indx} + \text{blockDim.x}]; \\
\text{dest}[\text{indx}] = \text{a0}; \\
\text{dst}[\text{indx} + \text{blockDim.x}] = \text{a1};
\]

Copy 4 floats /thread, need 25% occ
\[
\text{int}\ \text{indx} = \text{threadIdx.x} + 4 \times \text{block} \times \text{blockDim.x}; \\
\text{float a[4]; // in registers} \\
\text{for}(i=0;i<4;i++) \text{a[i]=src[indx+i*blockDim.x];} \\
\text{for}(i=0;i<4;i++) \text{dst[indx+i*blockDim.x]=a[i];}
\]
Incremental improvements to matrix multiply

• Baseline code found in the SDK (3.1, GTX 480)
• Follows V. Volkov [GTC10]
• Gets 137 Gflops / sec

```c
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```
Two outputs / thread

• 2 outputs, double the loads

```cpp
float Csub[2] = {0,0}; // array is allocated in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
    a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```
Two outputs / thread, part 2

- x2 flops and stores
- 341 Gflops/sec

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, k) * BS(k, tx);
    }
    __syncthreads();

    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub[0];
    C[c + wB * (ty+16) + tx] = Csub[1];
```
4 outputs / thread

```c
float Csub[4] = {0,0,0,0}; //array is in registers
for (int a = aBegin, b = bBegin; a <= aEnd;
    a += aStep, b += bStep)
{
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```
4 outputs / thread

- 427 Gflops/sec [w/8 output/thread → 485 Gflops/s)
- ×2 # registers
- 50% occupancy

```c
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+8, k) * BS(k, tx);
    Csub[2] += AS(ty+16, k) * BS(k, tx);
    Csub[3] += AS(ty+24, k) * BS(k, tx);
}
__syncthreads();
```

```c
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```
Vector length: 64 //stripmined into two warps by GPU
Registers: \( \mathbf{a}, \mathbf{c}[1:16] \) //each is 64-element vector
Shared memory: \( \mathbf{b}[16][16] \) //may include padding

Compute pointers in \( \mathbf{A}, \mathbf{B} \) and \( \mathbf{C} \) using thread ID
\( \mathbf{c}[1:16] = 0 \)
do
\( \mathbf{b}[1:16][1:16] = \) next \( 16 \times 16 \) block in \( \mathbf{B} \) or \( \mathbf{B}^T \)
local barrier //wait until \( \mathbf{b}[][] \) is written by all warps
unroll for \( i = 1 \) to \( 16 \) do
\( \mathbf{a} = \) next \( 64 \times 1 \) column of \( \mathbf{A} \)
\( \mathbf{c}[1] += \mathbf{a} \times \mathbf{b}[i][1] \) // rank-1 update of \( \mathbf{C} \)'s block
\( \mathbf{c}[2] += \mathbf{a} \times \mathbf{b}[i][2] \) // data parallelism = 1024
\( \mathbf{c}[3] += \mathbf{a} \times \mathbf{b}[i][3] \) // stripmined in software
... // into 16 operations
\( \mathbf{c}[16] += \mathbf{a} \times \mathbf{b}[i][16] \) // access to \( \mathbf{b}[][] \) is stride-1
endfor
local barrier //wait until done using \( \mathbf{b}[][] \)
update pointers in \( \mathbf{A} \) and \( \mathbf{B} \)
repeat until pointer in \( \mathbf{B} \) is out of range
Merge \( \mathbf{c}[1:16] \) with \( 64 \times 16 \) block of \( \mathbf{C} \) in memory

Figure 4: The structure of our matrix-matrix multiply routines.
SGEMM Code

```c
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y *16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;

    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
        #pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i] = B[i*ldb];
        B += 16;
        __syncthreads();

        #pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];
            c[1] += A[0]*bs[i][1];
            c[2] += A[0]*bs[i][2];
            c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];
            c[5] += A[0]*bs[i][5];
            c[6] += A[0]*bs[i][6];
            c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];
            c[9] += A[0]*bs[i][9];
            c[10] += A[0]*bs[i][10];
            c[12] += A[0]*bs[i][12];
            c[13] += A[0]*bs[i][13];
            c[14] += A[0]*bs[i][14];
            c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

- Compute pointers to the data
- Declare the on-chip storage
- Read next B’s block
- The bottleneck: Read A’s columns
- Do Rank-1 updates
- Store C’s block to memory
Data motion cost

- Communication performance is a major factor in determining the overall performance of an application

- The $\alpha - \beta$ model: $\alpha + \beta^{-1} \infty \ n$

  $n = \text{message length}$

  $\alpha = \text{message startup time}$

  $\beta \infty = \text{peak bandwidth (bytes / second)}$

<table>
<thead>
<tr>
<th>Machine</th>
<th>$\beta \infty \ (\text{Dev})$</th>
<th>H-D</th>
<th>D-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forge</td>
<td>103 GB/s</td>
<td>2.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Lilliput</td>
<td>73.6</td>
<td>3.3</td>
<td>2.8</td>
</tr>
<tr>
<td>CseClass01</td>
<td>122</td>
<td>3.4</td>
<td>2.7</td>
</tr>
<tr>
<td>CseClass04</td>
<td>56.1</td>
<td>5.2</td>
<td>4.1</td>
</tr>
</tbody>
</table>

As reported by bandwidthTest
Consequences of data motion cost

• Consider saxpy $z[i] = a*x[i]+y[i]$
• This is a bandwidth bound kernel
• Running time under the $\alpha-\beta$ model: $\alpha + \beta^{-1}_\infty \ n$
  \[ \alpha = 4 \ \mu s \]
  \[ \beta_\infty = 127 \ \text{GB/sec} \]
• Flop rate bounded by $(2n \ \text{flops}/12n \ \text{bytes})* \ \beta_\infty$
  • 27 Gflops/sec

• $N_{1/2}$ Half bandwidth point: $N \approx 42,000$
  • Matrix multiplication takes 4 $\mu s$
  • But the largest matrix that fits into memory is 1GB $\sim (16K)^2$
  • Consequence: saxpy takes constant time to run
Half power point

- We define the **half power point** \( n_{1/2} \) as the transfer size required to achieve \( \frac{1}{2} \beta_\infty \)

\[
\frac{1}{2} \beta^{-1}_\infty = n_{1/2} / T(n_{1/2}) \implies \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_\infty
\]

- In theory, this occurs when \( \alpha = \beta^{-1}_\infty n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_\infty \)

- Formula may not be accurate

![Graph showing N_{1/2} \approx 100KB](SDSC Blue Horizon)