Lecture 9

Performance Programming:
Bank Conflicts
Memory coalescing
Improved Matrix Multiply
Tree summation
Announcements

• A3 has been posted
• Next Tuesday’s lecture: be prepared to discuss the assigned paper in class: “Debunking to 100X GPU vs CPU myth …
Today’s lecture

• Projects
• Using Shared Memory – part II
• Memory coalescing
• Improvements to Matrix Multiplication
Projects

• **Multigrid (MPI or CUDA).** A multi-level method that uses a progression of meshes to accelerate the solution of equations

• **Communication avoiding matrix multiplication** (MPI or UPC)

• Conduct a *performance qualification* of Fermi. Evaluate cache behavior and other architectural features

• **Scalable 3D Fast Fourier Transform** (MPI or UPC)

• Particle simulation (MPI)
  
  www.cs.berkeley.edu/~ballard/cs267.sp11/hw2

• Performance comparison of OpenMP and pthreads

• Groups of 3 doing MPI projects can add communication overlap

• Make your choice by 2/24
  
  cseweb.ucsd.edu/classes/wi12/cse260-a/Projects/ProjectList.html
Recapping from last time
Shared Memory/Cache

- On-chip memory
- Per SM: Combination of L1 cache and shared memory
  - 16KB + 48 KB shared memory + L1
  - 48KB + 16 KB shared memory + L1
- L2 cache shared by all SMs
- Cache accesses to local or global memory, including temporary register spills
- Cache inclusion (L1 ⊆ L2?) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size
Improving locality in matrix multiply

- Naïve algorithm
  - Each thread loads all the data it needs, independently loads a row and column of input
  - Each input element loaded multiple times
  - Each thread computes 1 MAD + 2 loads + 1 store

- Blocked algorithm
  - Threads cooperate to load a block of A&B into on-chip shared memory
  - Each thread in the block performs the $ijk$ loop within shared memory
  - Each thread: $b$ mpy-adds + 1 load + 1 store
Performance of naïve (global mem) version

- Baseline [N=512]
  - Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
  - Forge, M2070 14×32 cores
  - 21 GF on 4 CPU cores (MPI), 25 Gflops for N=2K

<table>
<thead>
<tr>
<th>Gflops dp Lilliput</th>
<th>9.8</th>
<th>8.5</th>
<th>7.4</th>
<th>5.9</th>
<th>5.3</th>
<th>5.1</th>
<th>3.0</th>
<th>2.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>2×256</td>
<td>2×128</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>8×64</td>
<td>8×32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gflops sp Lilliput</th>
<th>8.6</th>
<th>7.7</th>
<th>6.2</th>
<th>4.6</th>
<th>3.9</th>
<th>3.5</th>
<th>2.0</th>
<th>1.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>2×256</td>
<td>2×128</td>
<td>2×32</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>8×64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gflops sp Forge dp</th>
<th>65 48 64 46</th>
<th>56 39 52 29</th>
<th>46 29</th>
<th>33 27</th>
<th>21</th>
<th>8.6</th>
<th>6.8</th>
<th>6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>2×128 2×256</td>
<td>2×64 4×128 4×64</td>
<td>4×32 2×32 8×64</td>
<td>16×32</td>
<td>32×16</td>
<td>32×8</td>
<td>32×4</td>
<td></td>
</tr>
</tbody>
</table>
An even faster matrix multiply

• Use shared memory to increase re-use

• Avoid thread divergence

• Memory Coalescing, avoid Bank Conflicts
  ● Next time
Using shared memory (uncoalesced glbl)

```c
__global__ void matMul( float* C, float* A, float* B, int N) {
    const unsigned int bx = BLOCK_X, by = BLOCK_Y;
    const unsigned int tx = threadIdx.x, ty = threadIdx.y;
    const unsigned int l = blockIdx.x*bx + tx, J = blockIdx.y*by + ty;
    const unsigned int gx = blockDim.x, gy = blockDim.y;

    __shared__ float a[BLOCK_X][BLOCK_Y], b[BLOCK_X][BLOCK_Y];
    if ((l < N) && (J < N)){
        float c = 0.0f;
        for (unsigned int k=0; k < gy; k++) {
            a[tx][ty] = A[ l*N+k*by+ty];
            b[ty][tx] = B[ J+N*(k*bx+tx)];
            __syncthreads(); // Synchronizes all threads in a block
            for (unsigned int kk=0; kk< bx; kk++)
                c += a[kk][tx]*b[kk][ty];
            __syncthreads(); // Avoids memory hazards
        }
    }
    C[l*N+J] = c;
}
```

Results – shared memory – C1060

- N=512, double precision
- Different thread geometries
- Baseline: 23 GFlops on 4 cores of Lilliput
  69 Gflops on 8 cores of Triton (double)

<table>
<thead>
<tr>
<th>Geometry</th>
<th>16 × 16</th>
<th>8 × 8</th>
<th>4 × 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncoalesced</td>
<td>9.2</td>
<td>8.9</td>
<td>8.2</td>
</tr>
<tr>
<td>Coalesced</td>
<td>125 (57)</td>
<td>53 (41)</td>
<td>12 (15)</td>
</tr>
<tr>
<td>Fermi (Forge)</td>
<td>174 (94)</td>
<td>100 (63)</td>
<td>-</td>
</tr>
</tbody>
</table>
Improving performance

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

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

Slow:
\[
I = \text{blockIdx.x} \times \text{bx} + \text{tx}; \\
J = \text{blockIdx.y} \times \text{by} + \text{ty};
\]

\[
\text{Slow:} \\
\text{a[tx][ty] = A[I*N+k*by+ty];} \\
\text{b[ty][tx] = B[J+N*(k*bx+tx)];}
\]

©2010 Scott B. Baden /CSE 260/ Fall '10
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 $n$ times than 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’s
- Each bank can service 1 address / cycle (bcast, 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

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
float a[idx] = 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?
  ```c
  int tid = threadIdx.x;
  shared[2*tid] = global[2*tid];
  shared[2*tid+1] = global[2*tid+1];
  ```
- To avoid conflicts:
  ```c
  shared[tid] = global[tid];
  shared[tid + blockDim.x] = global[tid + blockDim.x];
  ```
- Consider:
  ```c
  __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
  - Unlike 1.x, no bank conflicts here in the code example
- 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
Global Memory

• If accessed word > 4 bytes, warp’s memory request split into separate, independently issued 128-byte memory requests
• Non-atomic, concurrent writes within a warp: writer not defined
Global memory coalescing

- Global memory accesses in units of 32, 64, 128 B
- Consecutive addresses read quickly (K=0; Q=1)
- Certain non-sequential access patterns to global memory degrade performance $K \mod 16 \neq 0; Q \neq 1$
- Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)

\[
tid = blockIdx.x \times blockDim.x + threadIdx.x + K
\]
\[
\text{shared[tid]} = \text{global[tid]}
\]
\[
\text{int } tid = (blockIdx.x \times blockDim.x + threadIdx.x) \times Q
\]

Nvidia Corp.

Memory coalescing

- Simplest: addresses are contiguous across threads
- Accesses organized by half warps (16 threads)

<table>
<thead>
<tr>
<th></th>
<th>M₀₀</th>
<th>M₁₀</th>
<th>M₂₀</th>
<th>M₃₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₀₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₀₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₀₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₁₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₁₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₁₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₁₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₂₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₂₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₂₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₂₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₃₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₃₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₃₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₃₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Threads →

Tile iteration

Nvidia Corp.

Memory coalescing (compute capability ≥1.2)

- Find the segment containing the address request of the lowest numbered active thread
- Find all other active threads requesting in same segment
- Reduce transaction size (if possible)
- Mark the serviced threads as inactive
- Repeat until all threads in ½ warp are complete

1 transaction - 64B segment

2 transactions - 64B and 32B segments
Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns
  
  \[
  I = blockIdx.y \times by + ty; \\
  J = blockIdx.x \times bx + tx; \\
  int tx = threadIdx.x \\
  a[ty][tx] = A[I \times N + k \times by + tx] \\
  b[ty][tx] = B[J \times N + (k \times bx + ty)]
  \]

- Accesses by threads in a block along a column don’t coalesce
  
  \[
  I = blockIdx.x \times bx + tx; \\
  J = blockIdx.y \times by + ty; \\
  a[tx][ty] = A[I \times N + k \times by + ty] \\
  b[ty][tx] = B[J \times N + (k \times bx + tx)]
  \]
Volkov and Demmel’s SGEMM

Vector length: 64 //stripmined into two warps by GPU
Registers: a, c[1:16] //each is 64-element vector
Shared memory: b[16][16] //may include padding

Compute pointers in A, B and C using thread ID
\[ c[1:16] = 0 \]
do
\[ b[1:16][1:16] = \text{next 16x16 block in } B \text{ or } B^T \]
\textbf{local barrier} //wait until \( b[[]] \) is written by all warps
\textbf{unroll for } i = 1 \text{ to } 16 \textbf{ do}
\[ a = \text{next 64x1 column of } A \]
\[ c[1] += a*b[i][1] \quad \text{// rank-1 update of } C \text{'s block} \]
\[ c[2] += a*b[i][2] \quad \text{// data parallelism = 1024} \]
\[ c[3] += a*b[i][3] \quad \text{// stripmined in software} \]
\[ \ldots \quad \text{// into 16 operations} \]
\[ c[16] += a*b[i][16] \quad \text{// access to } b[[]] \text{ is stride-1} \]
\textbf{endfor}
\textbf{local barrier} //wait until done using \( b[[]] \)
update pointers in A and B
\textbf{repeat until } pointer in B is out of range
Merge \( c[1:16] \) with 64x16 block of \( C \) 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
    {
        __syncthreads();
        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];
    }
}
```

Thread divergence: reduction
Thread Divergence

• All the threads in a warp execute the same instruction

• Different control paths are serialized

• *Divergence* when a predicate is a function of the threadId
  
  \[
  \text{if (threadId < 2) \{ \}}
  \]

• No divergence if all follow the same path
  
  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]

• Consider reduction, e.g. summation \(\sum_i x_i\)
Divergence example

if (threadIdx >= 2)
    a=100;
else
    a=-100;

compare threadIdx,2

Mary Hall

Divergence example

if (threadIdx >= 2)  
a=100;
else  
a=-100;

Mary Hall

Divergence example

if (threadIdx >= 2)
a=100;
else
a=-100;

Mary Hall

Example – reduction – thread divergence

Thread 0 | Thread 2 | Thread 4 | Thread 6 | Thread 8 | Thread 10
---|---|---|---|---|---
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1+1 | 2+3 | 4+5 | 6+7 | 8+9 | 10+11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ int x[BSIZE];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }
    if (tid == 0) atomicAdd(total,x[tid]);
}
```

Reducing divergence and avoiding bank conflicts

Thread 0

The improved code

- No divergence until stride < 32
- All warps active when stride ≥ 32

```c
for (stride = blockDim.x/2;
    stride > 1;
    stride /= 2) {
    __syncthreads();
    if (tid < stride)
        x[tid] += x[tid + stride];
}
```

```c
for (stride = 1;
    stride < blockDim.x;
    stride *= 2) {
    __syncthreads();
    if (tid % (2*stride) == 0)
        x[tid] += x[tid + stride];
}
```
Predication on Fermi

• All instructions support predication in 2.x
• Condition code or *predicate* per thread:
  set to true or false
• Execute only if the predicate is true
  if \((x>1)\)
  \(y = 7;\)
  
  test = \((x>1)\)
  test: \(y=7\)
• Compiler replaces a branch instruction with predicated
  instructions only if the number of instructions
  controlled by branch condition is not too large
• If the compiler predicts too many divergent warps…..
  threshold = 7, else 4
Concurrency – Host & Device

- Nonbocking operations
  - Kernel launch
  - Device ↔ {Host, Device}
  - Async memory copies
- Multiple kernel invocations: certain capability 2.x devices
- CUDA Streams (§3.2.6.5), with limitations
  ```c
  cudaStream_t stream[2];
  for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
  for (int i = 0; i < 2; ++i)
    Kernel<<<100, 512, 0, stream[i]>>> ( ... );
  ```