Lecture 8

More on Coalescing
Avoiding Bank Conflicts
Further improvements to Matrix Multiply
Announcements

- Tuesday’s lecture on 2/11 will be moved to room 4140 from 6.30 PM to 7.50 PM
## Results from A1

<table>
<thead>
<tr>
<th>GiD</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATLAS</td>
<td>8.14</td>
<td>6.78</td>
</tr>
<tr>
<td>G-520-802-976</td>
<td>7.87</td>
<td>6.38</td>
</tr>
<tr>
<td>G-338-665</td>
<td>6.75</td>
<td>5.26</td>
</tr>
<tr>
<td>G-507-689</td>
<td>8.63</td>
<td>4.61</td>
</tr>
<tr>
<td>G-235-688</td>
<td>5.17</td>
<td>4.34</td>
</tr>
<tr>
<td>G-126-180-722</td>
<td>4.72</td>
<td>4.24</td>
</tr>
<tr>
<td>G-360-811-992</td>
<td>4.84</td>
<td>4.11</td>
</tr>
<tr>
<td>G-157-526-538</td>
<td>4.74</td>
<td>4.11</td>
</tr>
<tr>
<td>G-155-171-808</td>
<td>4.77</td>
<td>4.05</td>
</tr>
<tr>
<td>G-543-876</td>
<td>4.25</td>
<td>3.97</td>
</tr>
<tr>
<td>G-205-578</td>
<td>4.48</td>
<td>3.96</td>
</tr>
<tr>
<td>G-183-229-535</td>
<td>3.95</td>
<td>3.68</td>
</tr>
<tr>
<td>G-672</td>
<td>2.54</td>
<td>2.19</td>
</tr>
<tr>
<td>G-279-389-589</td>
<td>1.87</td>
<td>1.82</td>
</tr>
<tr>
<td>G-418-960</td>
<td>3.34</td>
<td>1.72</td>
</tr>
<tr>
<td>G-163-197</td>
<td>1.83</td>
<td>1.62</td>
</tr>
</tbody>
</table>
Assignment #1

- Blocking for cache will boost performance but a lot more is needed to approach ATLAS’ performance.

\[ R_\infty = 4 \times 2.33 = 9.32 \text{ Gflops} \]

\( \sim 87\% \) of peak
Today’s lecture

• Memory coalescing
• Avoiding bank conflicts
• Further Improvements to Matrix Multiply
Structure of blocked algorithm

- Threads cooperate to load blocks of A&B into shared memory
- Each thread in the block performs the $ijk$ loop within shared memory

```c
for (int k=0; k < N/BLK; k++) {
    Load blocks of A & B
    __syncthreads();
    for (int kk=0; kk< BLK; kk++)
        c += a[kk][tx]*b[kk][ty];
    __syncthreads();
}
C[I*N+J] = c;
```
Using shared memory (uncoalesced)

```c
__global__ void matMul( float* C, float* A, float* B, int N) {
    const unsigned int tx = threadIdx.x, ty = threadIdx.y;
    const unsigned int I = blockIdx.x*bx + tx, J = blockIdx.y*by + ty;
    __shared__ float a[BLK][BLK], b[BLK][BLK];
    float c = 0.0f;

    for (unsigned int k=0; k < N/BLK; k++) { // Sweep all blocks
        a[tx][ty] = A[I*N+k*BLK+ty]; // in block row/col
        b[ty][tx] = B[J+N*(k*BLK+tx)];
        __syncthreads(); // Synchronizes all threads in a block
        for (unsigned int kk=0; kk< BLK; kk++)
            c += a[kk][tx]*b[kk][ty];
        __syncthreads(); // Avoids memory hazards
    }
    C[I*N+J] = c;
}
```
Results – shared memory

- N=512, double precision
- Dirac (C2050)
- Shared memory didn’t improve performance

<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>40</td>
<td>39</td>
<td>14</td>
</tr>
<tr>
<td>Coalesced</td>
<td>93</td>
<td>66</td>
<td>14</td>
</tr>
</tbody>
</table>

Global memory variant

<table>
<thead>
<tr>
<th>Dirac (prefer SM) (prefer L1$)</th>
<th>57</th>
<th>50</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometries</td>
<td>64</td>
<td>58</td>
<td>42</td>
</tr>
</tbody>
</table>

Geometries 1×512 2×256 4×128
Coalescing with 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns

\[
I = \text{blockIdx.y} \times \text{BLK} + \text{ty}; \\
J = \text{blockIdx.x} \times \text{BLK} + \text{tx}; \\
\text{int tx = threadIdx.x} \\
a[\text{ty}][\text{tx}] = A[I \times \text{N} + k \times \text{BLK} + \text{tx}] \\
b[\text{ty}][\text{tx}] = B[J + \text{N} \times (k \times \text{BLK} + \text{ty})]
\]

- Accesses by threads in a block along a column don’t coalesce

\[
I = \text{blockIdx.x} \times \text{BLK} + \text{tx}; \\
J = \text{blockIdx.y} \times \text{BLK} + \text{ty}; \\
a[\text{tx}][\text{ty}] = A[I \times \text{N} + k \times \text{BLK} + \text{ty}] \\
b[\text{ty}][\text{tx}] = B[J + \text{N} \times (k \times \text{BLK} + \text{tx})]
\]
Coalescing with 2d arrays

- Accesses by threads in a block along a column don’t coalesce.

- All warps in a block access consecutive elements within a row as they step through neighboring columns.

```
I = blockIdx.x * BLK + tx;
J = blockIdx.y * BLK + ty;
int tx = threadIdx.x
a[tx][ty] = A[I * N + k * BLK + ty]
b[ty][tx] = B[J + N * (k * BLK + tx)]
```

David Kirk/NVIDIA & Wen-mei Hwu/UIUC

Scott B. Baden /CSE 260/ Winter 2014
Coalesced access improves performance

I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;

__shared__ float a[BLK][BLK], b[BLK][BLK];
float c = 0.0f;
for (k=0; k < N/BLK; k++){
    a[ty][tx] = A[I*N+k*BLK+tx];
    b[ty][tx] = B[J+N*(k*BLK+ty)];
    __syncthreads();
    for (kk=0; kk < BLK; kk++)
        c += a[ty][kk]*b[kk][tx];
    __syncthreads();
}
C[I*N+J] = c;

Uncoalesced:
I = blockIdx.x*BLK+ tx;
J = blockIdx.y*BLK+ ty;

Uncoalesced:
a[tx][ty] = A[I*N+k*BLK+ty];
b[ty][tx] = B[J+N*(k*BLK+tx)];

Linearized order in increasing address
Global memory coalescing

- Global memory accesses in units of 32, 64, 128 B
- Does not require sequential accesses by threads—only need to fall within the same 128B segment
- Accesses organized by warp
- Concurrent accesses by a warp’s reads coalesce into K transactions = # different cache lines (128B) covered by the accesses
- Access not cached in L1: 32B at a time (x4) in L2

shrd[threadIdx.x] = gbl[blockIdx.x*blockDim.x+threadIdx.x] YES!
shrd[threadIdx.x] = gbl[blockDim.x+blockIdx.x*threadIdx.x] NOT!

Nvidia Cuda Best Practices: Sec. 9.2.1

Scott B. Baden /CSE 260/ Winter 2014
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
• cudaMalloc() is guaranteed to be aligned to at least 256 bytes
Memory coalescing

- Simplest: addresses fall within the same 128B segment
- Accesses organized by warps (32 threads)
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 the warp are complete
- Handles permutations, too

1 transaction – 128B segment

2 transactions – 2 x 128B segments

Nvidia Cuda C Programming Guide: Appendix G.4.2
Today’s lecture

• Memory coalescing
• Avoiding bank conflicts
• Further Improvements to Matrix Multiply
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 (broadcast, too)
- Access to shared memory is fast unless…
  - 2 or more instructions in a warp access the same bank: we have a conflict
  - Exception: not if accesses to the same 32 bit word: broadcast
- For writes, only one thread writes, writer is undefined

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
a[idx] = a[idx]+1.0f;
```
Conflict free access

• Consider
  \[
  \textbf{__shared__} \text{ float } \text{shared}[256];
  \]
  \[
  \text{float } \text{foo} = \text{shared}[\text{base} + s \times \text{threadIdx.x}];
  \]

• If \(s\) has no common factors with the number of banks (32), then there are no conflicts (\(s\) is odd)
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];
  ```

A memory system with 4 banks.
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 entire warp
  - No conflicts if access to bytes in same 32 bit word
- For devices of compute capability 1.x
  - Number of banks = 16
  - Bandwidth is 32 bits per bank per clock cycle
  - Shared memory request for a warp is split in two half-warps
  - Decreased susceptibility to bank conflicts
  - 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*BLK+ ty;
J = blockIdx.x*BLK+ tx;

__shared__ float     a[BLK][BLK],   b[BLK][BLK];
if ((I < N) && (J < N)){
    float c = 0.0f;
    for (k=0; k < N/BLK; k++){
        a[ty][tx] = A[I*N+k*BLK+tx];
        b[ty][tx] = B[J+N*(k*BLK+ty)];
        __syncthreads();
        for (kk=0; kk< BLK; kk++)
            c += a[ty][kk]*b[kk][tx];  all access same bank: broadcast
        __syncthreads();
    }
    C[I*N+J] = c;
}

Slow:
I = blockIdx.x*BLK+ tx;
J = blockIdx.y*BLK+ ty;

__shared__ float     a[BLK][BLK],   b[BLK][BLK];
if ((I < N) && (J < N)){
    float c = 0.0f;
    for (k=0; k < N/BLK; k++){
        a[tx][ty] = A[I*N+k*BLK+ty];
        b[ty][tx] = B[J+N*(k*BLK+tx)];
        __syncthreads();
        for (kk=0; kk< BLK; kk++)
            c += a[ty][kk]*b[kk][tx];  all access same bank: broadcast
        __syncthreads();
    }
    C[I*N+J] = c;
Today’s lecture

• Memory coalescing
• Avoiding bank conflicts
• Further Improvements to Matrix Multiply
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
  \[ x = a + b; \quad // \sim 20 \text{ for floating point, } 500+ \text{ for memory} \]
  \[ y = a + c; \quad // \text{ independent (stall)} \]
  \[ z = x + d; \quad // \text{ dependent, wait} \]

• How many warps are needed to hide latency
  if minimum latency is 4 cycles / instruction?

  \[
  \text{# Parallelism (threads)} = \text{latency} \times \text{throughput} \quad T = \lambda \times p
  \]
  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
- 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;
}
```
Hiding memory latency

• Parallelism = latency × throughput

Arithmetic: 576 ops/SM = 18CP x 32/SM/CP
Memory: 150KB = ~500CP (1100 nsec) x 150 GB/sec

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

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

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

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

- Baseline code found in the SDK
- 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

```c
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

```c
#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();
```

```c
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+16) + tx] = Csub[1];
```
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];
```
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
\texttt{c[1:16] = 0}
do
\texttt{b[1:16][1:16] = next 16\times16 block in B or B^T}
l\texttt{ocal barrier} //wait until \texttt{b[][]} is written by all warps
unroll \texttt{for} \texttt{i = 1 to 16} \texttt{do}
\texttt{a = next 64\times1 column of A}
\texttt{c[1] += a*b[i][1]} // rank-1 update of C’s block
\texttt{c[2] += a*b[i][2]} // data parallelism = 1024
\texttt{c[3] += a*b[i][3]} // stripmined in software
\ldots
\texttt{c[16] += a*b[i][16]} // access to \texttt{b[][]} is stride-1
endfor
\texttt{local barrier} //wait until done using \texttt{b[][]}
update pointers in A and B
\texttt{repeat until} pointer in B is out of range
\texttt{Merge c[1:16] with 64\times16 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
  {
    #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 )
    {
    }
    __syncthreads();
  } while( B < Blast );
  for( int i = 0; i < 16; i++, C += ldc )
    C[0] = alpha*c[i] + beta*C[0];
}

Volkov and Demmel
```

Scott B. Baden / CSE 260/ Winter 2014
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>$\beta_{\infty}$ (Dev)</th>
<th>H-D</th>
<th>D-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirac</td>
<td>104 GB/s</td>
<td>5.8</td>
<td>5.6</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

Scott B. Baden /CSE 260/ Winter 2014
Half power point

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

  $$\frac{1}{2} \beta^{-1}_\infty = \frac{n_{1/2}}{T(n_{1/2})} \Rightarrow \beta^{-1}(n_{1/2}) = \frac{1}{2} \beta^{-1}_\infty$$

- 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 half power point](SDSC Blue Horizon)
Consequences of data motion cost

• Consider saxpy: \( z[i] = a\times x[i] + y[i] \)
  Performs 2 flops per 3 words of memory written and read
• This is a bandwidth bound kernel
  Running time \( \approx \alpha + (\text{Required Bandwidth})\times\beta^{-1}_{\infty} \)
  \( \alpha = 4\,\mu s, \quad \beta_{\infty} = 127\,\text{GB/sec} \)
• Flop rate bounded by
  \( (2n\,\text{flops}/12n\,\text{bytes}) \times 127\,\text{GB/sec} = 27\,\text{Gflops/sec} \)
• \( N_{1/2} \) Half bandwidth point: \( N \approx 42,000 \)
  Half the time is spent in the \( \alpha \) term: 8\( \mu \)s
• Saxpy is used in matrix linear algebra, with matrices
  • But matrix size \( N \) cannot exceed \( \sqrt{M/24} \)
    \( M \sim 1\,\text{GB} \Rightarrow N \approx 16,000 \)
  • Consequence: saxpy takes constant time to run for practical matrix sizes
Fin