Lecture 7

Overlap
Using Shared Memory
Performance programming the memory hierarchy
Announcements

• Mac Mini lab (APM 2402)
  ♦ Starts Tuesday
  ♦ Every Tues and Fri for the next 2 weeks

• Project proposals due on November 9 (revised date)
Projects

• Counts for 60% of your grade
• Complete in 3 weeks
• See the (growing) list of projects at cseweb.ucsd.edu/classes/fa12/cse260-b/Projects/ProjectList.html
• CUDA, MPI or CUDA + MPI
• Select your project from the list by 11/9
  • A limited number of self-proposed projects, requires a proposal
• Progress report: 11/21 (Weds)
• Final Report: 12/7 (Friday)
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

• Overlap
• Occupancy
• Improving Matrix Multiplication performance with shared memory
• Memory coalescing
• Avoiding bank conflicts
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

©2012 Scott B. Baden /CSE 260/ Fall 2012
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?
Instruction Issue on Fermi

• Each vector unit
  • 32 CUDA cores for integer and floating-point arithmetic
  • 4 special function units for Single Precision transcendentals
• 2 Warp schedulers: each scheduler issues: 1 (2) instructions for capability 2.0 (2.1)
• One scheduler in charge of odd warp IDs, the other even warp IDs
• Only 1 scheduler can issue a double-precision floating-point instruction at a time
• Warp scheduler can issue an instruction to ½ the CUDA cores in an SM
• Scheduler must issue the instruction over 2 clock cycles for an integer or floating-point arithmetic instruction
Dynamic behavior – resource utilization

- Each vector core (SM): 1024 thread slots and 8 block slots
- Hardware partitions slots into blocks at run time, accommodates different processing capacities
  - Macmini 320M: 6 SMs
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block
Today’s lecture

• Overlap
• **Occupancy**
• Improving Matrix Multiplication performance with shared memory
• Memory coalescing
• Avoiding bank conflicts
Occupancy

• A minimum number of warps needed to hide memory latency
• **Occupancy**: \( \frac{\text{# active warps}}{\text{max \# warps supported by vector unit}} \)
• Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads
• Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 \( [38*256 = 9728 < 16k] \)
  - # available registers is the limiting factor
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  - Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  - Register consumption
• Maximizing the occupancy may not maximize performance
Occupancy Calculator

Determining occupancy

• Recall the definition for occupancy
  \[ \frac{\text{# active warps}}{\text{max # warps supported by vector unit}} \]

• NVIDIA provides an occupancy calculator

• Determine resource usage from nvcc
  
  nvcc --ptxas-options=-v
  Used 10 registers, 2092+16 bytes smem
CUDA GPU Occupancy Calculator

1.) Select Compute Capability (click): 2.0

2.) Enter your resource usage:
   Threads Per Block 256
   Registers Per Thread 10
   Shared Memory Per Block (bytes) 2092

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
   Active Threads per Multiprocessor 1536
   Active Warps per Multiprocessor 48
   Active Thread Blocks per Multiprocessor 6
   Occupancy of each Multiprocessor 100%

Physical Limits for GPU Compute Capability: 2.0
   Threads per Warp 32
   Warps per Multiprocessor 48
   Threads per Multiprocessor 1536
   Thread Blocks per Multiprocessor 8
   Total # of 32-bit registers per Multiprocessor 32768
   Register allocation unit size 64
   Register allocation granularity warp
   Shared Memory per Multiprocessor (bytes) 49152
   Shared Memory Allocation unit size 128
   Warp allocation granularity (for register allocation) 0

Allocation Per Thread Block
   Warps 8
   Registers 2560
   Shared Memory 2176

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor Blocks
   Limited by Max Warps / Blocks per Multiprocessor 6
   Limited by Registers per Multiprocessor 12
   Limited by Shared Memory per Multiprocessor 22

Thread Block Limit Per Multiprocessor highlighted RED

Occupancy calculation with 16 x 16 threads
THESE SLIDES NEED WORK – RERUN!

Occupancy = # active warps per SM
               Maximum possible # active warps
Full occupancy

Varying Block Size

Varying Register Count

Varying Shared Memory Usage
8 x 8 thread blocks

2.) Enter your resource usage:
Threads Per Block
Registers Per Thread
Shared Memory Per Block (bytes)

Maximum Thread Blocks Per Multiprocessor
Limited by Max Warps / Blocks per Multiprocessor
Limited by Registers per Multiprocessor
Limited by Shared Memory per Multiprocessor
Thread Block Limit Per Multiprocessor highlighted

©2012 Scott B. Baden /CSE 260/ Fall 2012
17
Summary - Programming issues

• Branches serialize execution within a warp
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  ♦ Register consumption
  ♦ Scheduling: hide latency
• Shared memory and registers do not persist across kernel invocations
• Next time: using shared memory
Today’s lecture

- Overlap
- Occupancy
- **Improving Matrix Multiplication performance with shared memory**
- Memory coalescing
- Avoiding bank conflicts
Speeding up matrix multiply

• Use shared memory to increase re-use
• Avoid thread divergence
• Memory Coalescing, avoid Bank Conflicts
**Blocked Matrix Multiplication**

```plaintext
for i = 0 to N-1
    for j = 0 to N-1
        // load each block C[i,j] into cache, once :
        n^2
        // b = n/N = block size
        for k = 0 to N-1
            // load each block A[i,k] and B[k,j] N^3 times
            // = 2N^3 \times (n/N)^2 = 2Nn^2
            // write each block C[i,j] once :
            n^2
        
    
Total: (2*N+2)*n^2
```

©2012 Scott B. Baden /CSE 260/ Fall 2012 21
Naïve kernel implementation

- Each thread computes one element of C
  - Loads a row of matrix A
  - Loads a column of matrix B
  - Computes a dot product
- Every value of A and B is loaded N times from global memory
Naïve Kernel

```c
__global__ void matMul(DOUBLE* C, DOUBLE* A, DOUBLE* B) {
    int I = blockIdx.x*blockDim.x + threadIdx.x;
    int J = blockIdx.y*blockDim.y + threadIdx.y;
    int N = blockDim.y*gridDim.y; // Assume a square matrix
    if ((I < N) && (J < N)){
        float _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            float a = A[I * N + k];
            float b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}
```
Shared Memory/Cache

- On-chip local store per SM: shared memory + L1$
  - 16KB shared memory + 48 KB L1 cache
  - 48KB shared memory + 16 KB L1 cache
- All threads in a block share this on-chip memory
  - A collection of warps share a portion of the local store
- Cache local or global memory, including temporary register spills
- L2 cache shared by all vector units
  - Cache inclusion (L1 ⊆ L2) configurable at compile time: L1 & L2 or L1 only
- 128 byte cache line size
- Set the preference; initial 48KB Shared, 16KK L1
  cudaFuncSetCacheConfig( boundariesX, PREFERENCE )
  PREFERENCE = {cudaFuncCachePreferShared, cudaFuncCachePreferL1}
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 with shared memory
  - 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
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;
    }
}
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Results – shared memory

- N=512, double precision
- Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
- (Dirac) C2050
- 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 (38)</td>
<td>8.9</td>
<td>8.2</td>
</tr>
<tr>
<td>Coalesced</td>
<td>57 (93)</td>
<td>41</td>
<td>15</td>
</tr>
</tbody>
</table>

Global memory variant

<table>
<thead>
<tr>
<th>Gflops dp Lilliput Dirac (prefer SM) (prefer L1$)</th>
<th>9.8</th>
<th>8.5</th>
<th>7.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2050</td>
<td>57</td>
<td>53</td>
<td>48</td>
</tr>
<tr>
<td>C2050</td>
<td>62</td>
<td>56</td>
<td>58</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Lilliput Dirac</th>
<th>2×256</th>
<th>2×128</th>
<th>2×64</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2050</td>
<td>1×512</td>
<td>2×256</td>
<td>2×128</td>
<td></td>
</tr>
<tr>
<td>C2050</td>
<td>2×512</td>
<td>2×128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©2012 Scott B. Baden /CSE 260/ Fall 2012
Today’s lecture

- Overlap
- Occupancy
- Improving Matrix Multiplication performance with shared memory
- Memory coalescing
- Avoiding bank conflicts
Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?
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 \(K \ mod \ 16 \neq 0; \ Q\neq1\)
- Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)

\[
\begin{align*}
tid &= \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} + K \\
\text{shared}[tid] &= \text{global}[tid] \\
\text{int } tid &= (\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}) \times Q
\end{align*}
\]
Memory coalescing

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

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

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

  \[
  I = \text{blockIdx.x} \times \text{bx} + \text{tx}; \\
  J = \text{blockIdx.y} \times \text{by} + \text{ty}; \\
  \text{a[tx][ty] = A[I \times \text{N} + k \times \text{by} + \text{ty}]} \\
  \text{b[ty][tx] = B[J \times \text{N} + (k \times \text{bx} + \text{tx})]}
  \]
Coalesced access improve performance

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

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

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

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

©2012 Scott B. Baden /CSE 260/ Fall 2012
Today’s lecture

• Overlap
• Occupancy
• Improving Matrix Multiplication performance with shared memory
• Memory coalescing
• Avoiding bank conflicts
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 \textit{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

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

©2012 Scott B. Baden /CSE 260/ Fall 2012
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
    • Unlike 1.x, no bank conflicts in the given 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
Coalesced access and no bank conflicts

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

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