Lecture 7

Using Shared Memory

Performance programming and the memory hierarchy
Announcements
Assignment #1

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

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

\[ \sim 87\% \text{ of peak} \]
Today’s lecture

- 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**: \# active warps ÷ 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
  # active warps ÷ max # warps supported by vector unit

• NVIDIA provides an occupancy calculator

• Determine resource usage from nvcc for provided A2 code on Dirac (capability 2.0)
  nvcc --ptxas-options=-v
  Used 25 registers, 64 bytes cmem[0]
Occupancy calculation with 16 x 16 threads

Occupancy = \( \frac{\text{# active warps per SM}}{\text{Maximum possible # active warps}} \)

Physical Limits for GPU Compute Capability: 2.0

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps per Multiprocessor</td>
<td>48</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>1536</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Total # of 32-bit registers per Multiprocessor</td>
<td>32768</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>64</td>
</tr>
<tr>
<td>Register allocation granularity</td>
<td>warp</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>63</td>
</tr>
<tr>
<td>Shared Memory per Multiprocessor (bytes)</td>
<td>49152</td>
</tr>
<tr>
<td>Shared Memory Allocation unit size</td>
<td>128</td>
</tr>
<tr>
<td>Warp allocation granularity</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Thread Block Size</td>
<td>1024</td>
</tr>
</tbody>
</table>

Allocated Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block</th>
<th>Limit Per SM</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps (Threads Per Block / Threads Per Warp)</td>
<td>8</td>
<td>48</td>
<td>6</td>
</tr>
<tr>
<td>Registers (Warp limit per SM due to per-warp reg count)</td>
<td>8</td>
<td>38</td>
<td>4</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>0</td>
<td>49152</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

<table>
<thead>
<tr>
<th>Limit</th>
<th>Blocks/SM</th>
<th>* Warps/Block = Warps/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps or Max Blocks per Multiprocessor</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Limited by Registers per Multiprocessor</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Limited by Shared Memory per Multiprocessor</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Note: Occupancy limiter is shown in orange

CUDA GPU Occupancy Calculator

1.) Select Compute Capability (click): 2.0
1.b) Select Shared Memory Size Config (bytes) 49152

2.) Enter your resource usage:
- Threads Per Block: 256
- Registers Per Thread: 25
- Shared Memory Per Block (bytes): 0

3.) GPU Occupancy Data
- Active Threads per Multiprocessor: 1024
- Active Warps per Multiprocessor: 32
- Active Thread Blocks per Multiprocessor: 4
- Occupancy of each Multiprocessor: 67%

Allocated Resources = Allocatable Blocks Per SM

Scott B. Baden /CSE 260/ Winter 2014
Full occupancy

Impact of Varying Block Size

Impact of Varying Register Count Per Thread

My Block Size 64

My Register Count 25

Scott B. Baden /CSE 260/ Winter 2014
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: using shared memory
Today’s lecture

- 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
Shared Memory/Cache

- On-chip local store: part shared memory, part L1
  - 16KB shared memory + 48 KB L1 cache
  - 48KB shared memory + 16 KB L1 cache
  - 1 for each vector unit
  - All threads in a block share this on-chip memory
    - A collection of warps share a portion of the local store
- Cache accesses to local or global memory, including temporary register spills
- L2 cache shared by all vector units
- Cache inclusion (L1 ⊂ L2) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size
- Set the mode using `cudaFuncSetCacheConfig()
  `cudaFuncSetCacheConfig(boundariesX, PREFERENCE )
  PREFERENCE = {cudaFuncCachePreferShared, cudaFuncCachePreferL1}`
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
__global__ void
matMul(int N 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)){
        DOUBLE _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            DOUBLE a = A[I * N + k];
            DOUBLE b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}

for (unsigned int i = 0; i < N; i++)
    for (unsigned int j = 0; j < N; j++) {
        DOUBLE sum = 0;
        for (unsigned int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * N + j];
        C[i * N + j] = (DOUBLE) sum;
    }
Recall Blocked Matrix Multiplication

**N** blocks, **n**×**n** global matrix, **b**=**n**/**N**

for **i** = 0 to **N**-1
    for **j** = 0 to **N**-1
        // load each block **C**[**i**,**j**] into cache, once:
        // **b** = **n**/**N** = block size
        for **k** = 0 to **N**-1
            // load each block **A**[**i**,**k**] and **B**[**k**,**j**] **N**^3 times
            // = 2**N**^3 × (**n**/**N**)^2
            **C**[**i**,**j**] += **A**[**i**,**k**] * **B**[**k**,**j**] // do the matrix multiply
            // write each block **C**[**i**,**j**] once:
            **n**^2

**Total:**

(2*N+2)***n**^2
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 $1 \text{ MAD } + 2 \text{ loads } + 1 \text{ store}$

• 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: $b \text{ mpy-adds } + 1 \text{ load } + 1 \text{ store}$
Using shared memory (uncoalesced global)

```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
        }
        C[l*N+J] = c;
    }
}
```

Scott B. Baden /CSE 260/ Winter 2014
Results – shared memory

- N=512, double precision
- Dirac (C2050)

<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>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometries</th>
<th>1×512</th>
<th>2×256</th>
<th>4×128</th>
</tr>
</thead>
</table>
Today’s lecture

• 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
- 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

```c
shrd[threadIdx.x] = gbl[blockIdx.x*blockDim.x+threadIdx.x] YES!
shrd[threadIdx.x] = gbl[blockDim.x+ blockIdx.x*threadIdx.x] NOT!
```
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 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 the 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
Coalescing with 2d arrays

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

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

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

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

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

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

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