Lecture 4

CUDA Programming
Today’s lecture

• Projects
• Matrix Multiplication
• Using Shared Memory
Projects

- Counts for 65% of your grade
- Complete in 7 weeks
  - Proposal: in class presentation + writeup (1/27)
  - Progress report #1: presentation (2/10)
  - Progress report #2: writeup (2/26)
  - Final Presentation: 3/13*
  - Final Report: 3/16 (Monday) at 5pm
Project Proposals

- Due 1/22
  - What are the goals of your project? Are they realistic?
  - What are your hypotheses?
  - What is your experimental method for proving or disproving your hypotheses?
  - What experimental result(s) do you need to demonstrate?
  - What would be the significance of those results?
  - What code will you need to implement? What software packages or previously written software will use?
  - A tentative division of labor among the team members
  - A preliminary list of milestones—with completion dates
Projects!

• CUDA+X, X= Multithreaded [+ MPI]
• Multigrid
  - Hybrid: GPU + CPU, also parallelize with MPI
• Communication avoiding matrix multiplication (CUDA + MPI)
• Option to add communication overlap
• Dynamic parallelism on host + device
• More information coming: watch the web site this weekend
• Propose your own
Overview of Kepler GK110

Multicore CPU

Control

Hyper-Q Execution Manager

Warp Scheduler
Dispatch
Single Precision

Warp Scheduler
Dispatch
Double Precision

Warp Scheduler
Dispatch
Single Precision

Warp Scheduler
Dispatch
Double Precision

SMX14

SMX1

SMX0

User Selectable Hardware/Software Cache

Level 2 Hardware Cache

DMA

Stream Optimized Device Memory

Latency Optimized Host Memory

©2013 The Portland Group, Inc.
Nvidia
**Execution Configurations**

- Grid ⊃ Block ⊃ Thread
- Expressed with *configuration variables*

```plaintext
__global__ void Kernel (...);

dim3 DimGrid(2,3);    // 6 thread blocks

dim3 DimBlock(3,5,1); // 15 threads /block

Kernel<<< DimGrid, DimBlock, >>>(...);
```

David Kirk / NVIDIA & Wen-mei Hwu / UIUC
Naïve implementation of matrix multiply

• 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

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC

Scott B. Baden / CSE 262 / UCSD, Wi'15
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++) {
            double a = A[I * N + k];
            double b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}
```
Thread mapping

```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;
    if ((I < N) && (J < N)){
        float _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;
    }
}
```
Performance

- \( N=512 \), double precision
- Stampede, K20, 2.0 GHz Intel Xeon E5-2680 0 @ 2.70GHz peak 21.6 GF / core

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 128</td>
<td>64</td>
</tr>
<tr>
<td>2 x 128</td>
<td>76</td>
</tr>
<tr>
<td>4 x 64</td>
<td>49</td>
</tr>
<tr>
<td>16 x 16</td>
<td>16</td>
</tr>
<tr>
<td>1 x 512</td>
<td>63</td>
</tr>
<tr>
<td>2 x 256</td>
<td>78</td>
</tr>
<tr>
<td>4 x 128</td>
<td>50</td>
</tr>
<tr>
<td>1 x 1024</td>
<td>62</td>
</tr>
<tr>
<td>2 x 512</td>
<td>79</td>
</tr>
<tr>
<td>4 x 256</td>
<td>51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># cores</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>160</td>
</tr>
</tbody>
</table>
Speeding up matrix multiply

• Use shared memory to increase re-use
• Avoid thread divergence
• Memory Coalescing, avoid Bank Conflicts
Kepler’s Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Can partition the on-chip Shared memory L1 cache
  \( \{ \frac{3}{4} + \frac{1}{4} \} \)
  \( \{ \frac{3}{4} + \frac{1}{4} \} \)
  \( \{ \frac{1}{2} + \frac{1}{2} \} \)
  - Set the mode using `cudaFuncSetCacheConfig()`
- L2 Cache (768 KB)
Recall Blocked Matrix Multiplication

\( N \) blocks, \( n \times n \) global matrix, \( b = n/N \)

for \( i = 0 \) to \( N-1 \)

\[ \text{for } j = 0 \text{ to } N-1 \]

\[ \text{// load each block } C[i,j] \text{ into cache, once : } n^2 \]

\[ \text{// } b = n/N = \text{ block size} \]

for \( k = 0 \) to \( N-1 \)

\[ \text{// load each block } A[i,k] \text{ and } B[k,j] \text{ } N^3 \text{ times} \]

\[ \text{// } = 2N^3 \times (n/N)^2 = 2Nn^2 \]

\[ C[i,j] += A[i,k] \times B[k,j] \text{ // do the matrix multiply} \]

\[ \text{// write each block } C[i,j] \text{ once : } n^2 \]

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 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
Structure of blocked algorithm

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

```
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[bx][by], b[bx][by];
    float c = 0.0f;

    for (unsigned int k=0; k < blockDim.y; k++) {  // Sweep all blocks
        a[tx][ty] = A[I*N+k*by+ty];                // in block row/col
        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[I*N+J] = c;
}
```
Results – shared memory

- N=512, double precision
- Shared memory didn’t improve performance: 73GF
- What happened?
- Hint: a warp benefits from accessing a contiguous aligned region of 128 or 256 bytes
Global memory coalescing

- Accesses organized by warp
- 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
- Concurrent accesses by a warp’s reads coalesce into K transactions = # different cache lines (128B) covered by the accesses
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)
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

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

73 $\rightarrow$ 161 GFLOPS

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

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

__shared__ float a[bx][by], b[bx][by];
float c = 0.0f;
for (k=0; k < gridDim.y; 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;
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
  ```
  __shared__ float shared[256];
  float foo = shared[base + s * 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];
  ```
Shared memory design

• Successive 32-bit words assigned to successive banks
• For devices of compute capability 3.x [Kepler]
  • Number of banks = 32
  • Bandwidth is 8 bytes per bank per clock per SMX
  • 256 bytes per clock per SMX
Coalesced code incurs no bank conflicts

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

\[ \_\_\text{shared}\_\_ \text{float} \quad a[\text{bx}][\text{by}], \quad b[\text{bx}][\text{by}]; \]
\[ \text{if } ( (I < N) \&\& (J < N)) \{ \]
\[ \text{float } c = 0.0f; \]
\[ \text{for } (k=0; \ k < \text{gridDim.y}; \ k++) \{ \]
\[ \quad a[\text{ty}][\text{tx}] = A[I*N+k*\text{by}+\text{tx}]; \]
\[ \quad b[\text{ty}][\text{tx}] = B[J+N*(k*\text{bx}+\text{ty})]; \]
\[ \_\_\text{syncthreads}(); \]
\[ \quad \text{for } (kk=0; \ kk < \text{bx}; \ kk++) \]
\[ \quad \quad c += a[\text{ty}][kk]*b[\text{kk}][\text{tx}]; \quad \text{all access same bank: broadcast} \]
\[ \_\_\text{syncthreads}(); \]
\[ \} \]
\[ C[I*N+J] = c; \]