Lecture 8

Matrix Multiplication
Using Shared Memory
Announcements

• A2 Due
• Friday’s makeup lecture 4:00 to 5:20pm in EBU3B 2154
• Next Tuesday’s lecture: be prepared to discuss the assigned paper in class: “Debunking to 100X GPU vs CPU myth …
Project Proposals

• Due 2/24
  ♦ 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
Today’s lecture

• Matrix Multiplication with Global Memory
• Using Shared Memory – part I
Recapping from last time
Many Multithreaded Vector Units

<table>
<thead>
<tr>
<th>GPU name</th>
<th>GeForce GTX280</th>
<th>GeForce 980GTX</th>
<th>GeForce 8800GTX</th>
<th>GeForce 8600GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td># of vector cores</td>
<td>30</td>
<td>16</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>core clock, GHz</td>
<td>1.30</td>
<td>1.67</td>
<td>1.35</td>
<td>1.45</td>
</tr>
<tr>
<td>registers/core</td>
<td>64KB</td>
<td>32KB</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>smem/core</td>
<td>16KB</td>
<td>16KB</td>
<td>16KB</td>
<td>16KB</td>
</tr>
<tr>
<td>memory bus, GHz</td>
<td>1.1</td>
<td>1.1</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>memory bus, pins</td>
<td>512</td>
<td>256</td>
<td>384</td>
<td>128</td>
</tr>
<tr>
<td>bandwidth, GB/s</td>
<td>141</td>
<td>70</td>
<td>86</td>
<td>32</td>
</tr>
<tr>
<td>memory amount</td>
<td>1GB</td>
<td>512MB</td>
<td>768MB</td>
<td>256MB</td>
</tr>
<tr>
<td>SP, peak Gflop/s</td>
<td>624</td>
<td>429</td>
<td>346</td>
<td>93</td>
</tr>
<tr>
<td>SP, peak per core</td>
<td>21</td>
<td>27</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>SP, flops/word</td>
<td>18</td>
<td>25</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>DP, peak Gflop/s</td>
<td>78</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>DP, flops/word</td>
<td>4.4</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 1: The list of the GPUs used in this study. SP is single precision and DP is double precision. Smem is shared memory. Peak flop rates are shown for multiply and add operations. Flops/word is the ratio of peak Gflop/s rate to pin-memory bandwidth in words.

Warp scheduling on Fermi

- Assign threads to an SM in units of a thread block
- Hardware is free to assign blocks to any SM, multiple blocks per SM
- Blocks are divided into warps of 32 (SIMD) threads, a schedulable unit
  - Dynamic instruction reordering
  - All threads in a Warp execute the same instruction
  - Multiple warps simultaneously active, hiding data transfer delays
  - All registers in all the warps are available, 0 overhead scheduling
**Occupancy**

- A minimum number of warps needed to hide memory latency
- **Occupancy**: \( \frac{\# \text{ active warps}}{\text{max} \# \text{ warps supported by vector unit}} \)
- 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
Data motion cost

- Communication performance is a major factor in determining the overall performance of an application

- The $\alpha - \beta$ model: $\alpha + \beta^{-1} \infty \ n$
  
  $n$ = message length
  
  $\alpha$ = message startup time
  
  $\beta_{\infty}$ = 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>Forge</td>
<td>103 GB/s</td>
<td>2.3</td>
<td>1.3</td>
</tr>
<tr>
<td>Lilliput</td>
<td>73.6</td>
<td>3.3</td>
<td>2.8</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

Consequences of data motion cost

• Consider saxpy \( z[i] = a \times x[i] + y[i] \)

• This is a bandwidth bound kernel

• Running time under the \( \alpha-\beta \) model: \( \alpha + \beta^{-1} \rightarrow \infty \) \( n \)
  \( \alpha = 4 \mu s \)
  \( \beta_{\infty} = 127 \text{ GB/sec} \)

• Flop rate bounded by (2n flops/12n bytes)* \( \beta_{\infty} \)
  • (1/6) flops per byte of bandwidth: 27 Gflops/sec

• \( N_{1/2} \) half power (bandwidth) point: \( N \approx 42,000 \)
  • The transfer size required to achieve \( \frac{1}{2} \beta_{\infty} \)
  • Matrix multiplication takes 4 \( \mu s \)
  • But the largest matrix that fits into memory is 1GB \( \sim (16K)^2 \)
  • Consequence: saxpy takes constant time to run
Half power point

- We define the \textit{half power point} $n_{1/2}$ as the transfer size required to achieve $\frac{1}{2} \beta_\infty$
  \[ \frac{1}{2} \beta_{-1\infty} = n_{1/2} / T(n_{1/2}) \Rightarrow \beta_{-1}(n_{1/2}) = \frac{1}{2} \beta_{-1\infty} \]
- In theory, this occurs when $\alpha = \beta_{-1\infty} n_{1/2} \Rightarrow n_{1/2} = \alpha \beta_\infty$
- Formula may not be accurate

\[ N_{1/2} \approx 100 \text{KB} \]

(SDSC Blue Horizon)

Latency

• Instructions waits on dependencies
  \( x = a + b; \quad \text{// } \sim 20 \)
  \( y = a + c; \quad \text{// independent} \)
  \( \text{(stall)} \)
  \( z = x + d; \quad \text{// dependent} \)

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

• Latencies, can vary but 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

• Measuring latencies
  - \( a = a*b+c, \text{ unrolled, 1 scalar thread on entire device} \)
  - No overflow
  - Hardware not optimized for special values 0,1
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
• See Volkov’s presentation:
  http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
• On the GF104, we must use ILP to go beyond 66% of peak
  • 48 cores/SM, half warp issues at a time
  • But we have only 2 schedulers
  • We must issue 2 instructions per warp in the same cycle

```
#pragma unroll UNROLL
for( int i = 0; i < N_ITERATIONS; i++ ){
  a = a*b+c;
  d = d * b + c;
}
```
Matrix Multiplication
(code in $PUB/Examples/CUDA/MM)$
Naïve Host Code

// “ijk” kernel
for i := 0 to n-1
  for j := 0 to n-1
    for k := 0 to n-1
      C[i,j] += A[i,k] * B[k,j]

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;
  }
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

___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;
}
}
CUDA code on the host side

```c
unsigned int n2 = N*N*sizeof(DOUBLE);
DOUBLE *h_A = (DOUBLE*) malloc(n2);
DOUBLE *h_B = (DOUBLE*) malloc(n2);
// Check that allocations went OK
assert(h_A); assert(h_B);

genMatrix(h_A, N, N); genMatrix(h_B, N, N); // Initialize matrices

DOUBLE *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, n2); ... &d_A ... &d_B
checkCUDAError("Error allocating device memory arrays");

// copy host memory to device
cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);
checkCUDAError("Error copying data to device");
```
// setup execution configurations
  dim3 threads(ntx, nty, 1);       // ntx & nty are user input
  dim3 grid(n / threads.x, N / threads.y);

  // launch the kernel
  matMul<<< grid, threads >>>(d_C, d_A, d_B);

  // retrieve result
  cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);
  checkCUDAError("Unable to retrieve result from device");

  // Free device storage
  assert(cudaSuccess == cudaFree(d_A));
  assert(cudaSuccess == cudaFree(d_B));
  assert(cudaSuccess == cudaFree(d_C));
Configuration variables

- Types to manage thread geometries
- \texttt{dim3 gridDim, blockDim}
  - Dimensions of the grid in blocks
    (\texttt{gridDim.z} not used)
  - Dimensions of a thread block in threads
- \texttt{dim3 blockIdx, threadIdx;}
  - Block index within the grid
  - Thread index within the block

\begin{verbatim}
__global__ void KernelFunc(...);
dim3 DimGrid(40, 30);    // 1200 thread blocks
dim3 DimBlock(4, 8, 16);   // 512 threads per block
Kernel<<< DimGrid, DimBlock, >>>(...);
\end{verbatim}
**Performance**

- **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>2×64</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 Forge dp</th>
<th>65</th>
<th>64</th>
<th>56</th>
<th>52</th>
<th>29</th>
<th>46</th>
<th>21</th>
<th>8.6</th>
<th>6.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>2×128</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>2×32</td>
<td>16×32</td>
<td>32×16</td>
<td>32×8</td>
</tr>
</tbody>
</table>

Memory Hierarchy

<table>
<thead>
<tr>
<th>Name</th>
<th>Latency (cycles)</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Local</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Constant</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
Shared Memory/Cache

- On-chip local store: pshared memory, partially 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 \texttt{cudaFuncSetCacheConfig()}
  \texttt{cudaFuncSetCacheConfig( boundariesX, PREFERENCE )}
  \texttt{PREFERENCE = \{cudaFuncCachePreferShared, cudaFuncCachePreferL1\} }
A better matrix multiply

• Use shared memory to increase re-use

• Avoid thread divergence

• Memory Coalescing, avoid Bank Conflicts
  - Next time
Improving locality

• 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
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 I = 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 ((I < N) && (J < N)) {
        float c = 0.0f;
        for (unsigned int k=0; k < gy; k++) {
            a[tx][ty] = A[ I*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[I*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>
</tbody>
</table>
Occupancy Calculator

Determining occupancy

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

• Maximizing the occupancy doesn’t always maximize performance

• NVIDIA provides an occupancy calculator

• Determine resource usage from `nvcc`
  ```
  nvcc --ptxas-options=-v
  Used 10 registers, 2092+16 bytes smem
  ```
Physical Limits for GPU:

1.3 Threads / Warp
32 Warps / Multiprocessor
32 Threads / Multiprocessor
1024 Thread Blocks / Multiprocessor
8 Limited by Max Warps / Multiprocessor
16384 Total # of 32-bit registers / Multiprocessor
512 Limited by Registers / Multiprocessor
16384 Limited by Shared Memory / Multiprocessor
16384 (bytes)
512 (bytes)
2 Warp allocation granularity (for register allocation)

CUDA GPU Occupancy Calculator

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

1.) Select Compute Capability (click):
1.3

2.) Enter your resource usage:

| Threads Per Block | 256 |
| Registers Per Thread | 10 |
| Shared Memory Per Block (bytes) | 2092 |

3.) GPU Occupancy Data is displayed here and in the graphs:

| Active Threads per Multiprocessor | 1024 |
| Active Warps per Multiprocessor | 32 |
| Active Thread Blocks per Multiprocessor | 4 |
| Occupancy of each Multiprocessor | 100% |
| Allocation Per Thread Block | |
| Warps | 8 |
| Registers | 2560 |
| Shared Memory | 2560 |

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor

| Blocks |
| Limited by Max Warps / Multiprocessor | 4 |
| Limited by Registers / Multiprocessor | 6 |
| Limited by Shared Memory / Multiprocessor | 6 |

Thread Block Limit Per Multiprocessor highlighted

Occupancy calculation with 16 x 16 threads

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: 64
- Registers Per Thread: 10
- Shared Memory Per Block (bytes): 2092

Maximum Thread Blocks Per Multiprocessor
- Limited by Max Warps / Multiprocessor: 8
- Limited by Registers / Multiprocessor: 16
- Limited by Shared Memory / Multiprocessor: 6

Thread Block Limit Per Multiprocessor highlighted RED
