Lecture 5

Performance Programming with CUDA
Announcements
Today’s lecture

• Matrix multiplication
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
Constraints

- **SM**
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps
- All threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization
- Grid: 1 or 2-dimensional (64K-1)
- Blocks – 1, 2, or 3-dimensional
  - ≤ 512 threads
  - Max dimensions: 512, 512, 64
  - Registers subdivided over threads
  - Synchronization among all threads in the block
- Synchronization only within block
Occupancy

- A minimum number of warps needed to hide memory latency
- Varying the block size
  - 8x8: 64 threads/block, max 8 blocks, only 512 threads/SM
  - 16x16: 3 blocks; can we hide the latency?
- Consider an application using 32 registers (4 bytes floats) per thread
- Each SM gets 512 registers, 64 per core, how many threads/SM?
“Cartoon:” how to obtain good speedups

• Avoid algorithms that present intrinsic barriers to utilizing the hardware
• Hide latency of host ↔ device memory transfers
• Reduce global memory accesses: use fast on-chip memories
• Manage occupancy
  ‣ Registers and shared memory
• Coalesced memory transfers
• Avoid costly branches, or render them harmless
Matrix Multiplication
(code in $PUB/Examples/CUDA/MM)$
Naïve Host Code

• “ijk” kernel

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;
    }
Square Matrix Multiplication

- \( C = A \times B \)
  
  Size: \( N \times N \)

- Naïve
  
  - Each thread handles one element of \( P \)
  
  - \( A \) & \( B \) loaded \( N \) times from global mem
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

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
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;
    }
}
```
N = blockDim.y * gridDim.y;
I = blockIdx.x * blockDim.x + threadIdx.x;
J = blockIdx.y * blockDim.y + threadIdx.y;
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;
}
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));
Performance

• N=512, double precision
• Lilliput, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
  ‣ 21 GF on 4 cores (MPI)
  ‣ Increases to 23, 25 Gflops for N=1024, 2048

<table>
<thead>
<tr>
<th>Gflops</th>
<th>9.8</th>
<th>8.5</th>
<th>7.4</th>
<th>5.8</th>
<th>5.3</th>
<th>5.0</th>
<th>3.0</th>
<th>1.3</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>32×4</td>
</tr>
</tbody>
</table>
A better matrix multiply

- Use shared memory to increase re-use
- Avoid thread divergence
- Memory Coalescing, avoid Bank Conflicts
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 $1 \text{ MAD} + 2 \text{ loads} + 1 \text{ store}$

• 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: $b \text{ mpy-adds} + 1 \text{ load} + 1 \text{ store}$
Results – shared memory

• N=512, single and 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>
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;
}
```
Shared memory bank conflicts

- Access to shared memory as fast as registers unless...
- If 2 or more instructions in a 1/2 warp access different banks: we have a conflict
- Exception: if all access the same bank: broadcast
- Each bank can service 1 address / cycle (bcast, too)

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

```
tid = blockIdx.x*blockDim.x+threadIdx.x + K
shared[tid] = global[tid]
int tid = (blockIdx.x*blockDim.x+ threadIdx.x)*Q
```

© 2011 Scott B. Baden / CSE 262 / Spring 2011
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

Nvidia Cuda C Programming Guide: Appendix G.3.2.2
Coalescing with 2d arrays

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

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

- Accesses by threads in a block along a column don’t coalesce.
Processing rate and effective bandwidth

Effective bandwidth = \( \frac{N_{\text{read}} + N_{\text{write}}}{\text{time}} \)

- Running time for Matrix Multiplication, \( N=512, b\{x,y\}=(16,16) \): 0.02147 sec
- Effective bandwidth = \( 3 \times 512^3 / 0.02097 = 18.7 \) GBytes/sec
- Peak bandwidth for C1060 102 GBytes / sec
Dynamic behavior – resource utilization

- Dynamic partitioning → underutilized resources
- Matrix multiply uses 13 reg/thread
- Blocked algorithm (single precision)
  - 16×16 blocks:
    - 3328 registers → 4 blocks (13312/16k)
  - 1024 thread limit → 4 blocks (125 GF)
  - 2080+16 bytes smem ((user + compiler)
  - 8×8 blocks:
    - 832 registers → 19 blocks
  - 8 block limit → 512 threads
  - Reduced performance: 125 → 53 GF
  - 544+16 bytes smem

NVIDIA

© 2011 Scott B. Baden / CSE 262 / Spring 2011
Reduction
Warp scheduling

• Blocks are divided into warps of 32 (SIMD) threads which are ….
• Subdivided into schedulable units: 16 threads (→ 32 on Fermi)
• Warps are scheduled with zero overhead in hardware
• Scheduler finds an eligible warp: all operands are ready
  ‣ Scoreboarding
  ‣ Priorities
• Branches serialize execution within a warp
Thread Divergence

• Blocks are divided into warps of 32 threads
• All the threads in a warp execute the same instruction
• Different control paths are serialized
• Divergence when predicate is a function of the threadId
  if (threadId < 2) { }
• No divergence if all follow the same path
  if (threadId / WARP_SIZE < 2) { }
• Consider reduction, e.g. summation $\sum_i x_i$
A naïve reduction

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 2</th>
<th>Thread 4</th>
<th>Thread 6</th>
<th>Thread 8</th>
<th>Thread 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
</tr>
<tr>
<td>0...3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..7</td>
<td></td>
<td></td>
<td></td>
<td>8..15</td>
<td></td>
</tr>
</tbody>
</table>

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

© 2011 Scott B. Baden / CSE 262 / Spring 2011
The naïve code

```c
__global__ void reduce(int *input, unsigned int N, int *total){
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ int x[BSIZE];
    x[tid] = (i<N) ? input[i] : 0;
    __syncthreads();

    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        __syncthreads();
        if (tid % (2*stride) == 0)
            x[tid] += x[tid + stride];
    }

    if (tid == 0) atomicAdd(total,x[tid]);
}
```
Reducing divergence and avoiding bank conflicts

Thread 0
The Code

```c
unsigned int tid = threadIdx.x;
__shared__ int x[BSIZE];

...

for (unsigned int s = blockDim.x/2; s>1; s /= 2) {
    __syncthreads();
    if (tid < s)
        x[tid] += x[tid + s];
}
```
Fin