Lecture 4

Performance Programming with CUDA
Announcements

• GPU access
  ‣ lilliput: Tesla C1060 (4 devices)
  ‣ cseclass0{1,2}
    Fermi GTX 570 (1 device each)

• MPI
  ‣ Trestles @ SDSC
  ‣ Kraken @ NICS
Assignment

• Starting from a basic (unoptimized) CUDA implementation
• Optimize matrix multiplication for the GPU
  ‣ Fermi and 200 series
• Next, implement matrix multiplication in MPI
  ‣ Lilliput →Trestles, if time Kraken
• You’ll be given an efficient (blocked for cache) multicore implementation that uses the Intel Math Kernel Library
• Goals
  ‣ Understand differences between Fermi and 200 series GPUs
  ‣ Comparative performance across platforms: how many MPI cores needed to deliver comparable performance to the GPU?
Today’s lecture

• Performance programming with CUDA
Recapping from last time

- Hierarchical thread model
- Fine grained threaded kernels operating on long vectors
- Fast on-chip memory
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
  - Grid ⊇ Block ⊇ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
- Thread Blocks
  - Subdivide a global index domain
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
- Threads are lightweight
  - Assigned to SM in units of blocks
  - Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

© 2011 Scott B. Baden / CSE 262 / Spring 2011
Threads, blocks, grids and data

- All threads execute same instruction (SIMT)
- A block may have a different number of dimensions (1d, 2d or 3d) than a grid (1d/2d)
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory locations
  - $\Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3$
  - $\Theta(\Pi_i), \forall \Pi_i \in \Pi$
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
Measuring performance

- Two ways
  - Use Cuda events/elapsed time (`#ifdef CUDA_TIMER`)
  - Use an ordinary timer, e.g. `gettimeofday()`
- See `incrArray`
- Note that kernel invocation is asynchronous

```c
cudaThreadSynchronize();
double t_device_compute = -getTime();
    COMPUTE
cudaThreadSynchronize();
t_device_compute += getTime();
```
Results with Increment

• The provided increment array benchmark
  ‣ Report performance on Lilliput for various values of N, # timesteps, and thread block sizes
  ‣ What settings minimized the running time per point?
  ‣ Compare performance with the host
• Change the increment function to the sin() function, scaling the input array to cover the range [0:2π)
  ‣ Why are the results different on the host and device?
  ‣ Comment out the verification code and report performance
  ‣ Add a loop to repeat the computation several times, account for what you observe
Experiments - increment benchmark

• Total time: timing taken from the host, includes copying data to the device
• Device only: time taken on device only

<table>
<thead>
<tr>
<th>Reps</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>8.5</td>
<td>83</td>
<td>830</td>
<td>8300</td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>100</td>
<td>850</td>
<td>8300</td>
</tr>
<tr>
<td>Host</td>
<td>77</td>
<td>770</td>
<td>7700</td>
<td></td>
</tr>
<tr>
<td>a[i] = 1 + sin(a[i]) : Device</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[i] = 1 + sinf(a[i]) : Host</td>
<td>16</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>23,000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Device Speedup

• How much of an improvement did our accelerated variant obtain over the traditional implementation?

• *Speedup, S*

Running time of the fastest program on conventional processors

Running time of the accelerated program

• Baseline: a multithreaded program
CUDA Error Handling

- Cuda can silently fail, you may observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions

Beware that the last error can be cleared by successive kernel calls, so check frequently

```c
assert(cudaSuccess == cudaMalloc(..);
printf("Cuda error: %s\n", cudaGetErrorString(cudaGetLastError()));
```

- Also see `checkCUDALError()` in `utils.cu` (incrArr)
- What about asynchronous calls?
Getting information about the binary

- Compiler will report a kernel’s register usage along with that of local, shared and constant memory
- Shared memory includes declared variables and system allocated data [function parameters]

```
--ptxas-options=-v

incrementArrays (float *a, int N)
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;
```

```
ptxas info : Compiling entry function
'_Z22incrementArrayOnDevicePfiif' for 'sm_13'
ptxas info : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1]
```
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
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block

Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC
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
Running times of CUDA instructions

• To execute an instruction for all threads of a warp, the warp scheduler must issue the instruction over
  ‣ 4 clock cycles for an integer or single-precision floating-point arithmetic instruction [throughput = 8]
  ‣ 32 cycles for double-precision [1]
  ‣ 16 cycles for single-precision floating-point transcendental [2]
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?
Programming issues

• Branches serialize execution within a warp
• Registers dynamically partitioned across each block in a Streaming Multiprocessor
• Bound to and only accessible from their thread until the block finishes execution
• 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
“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
How hard is it to get good performance?

• Simplified processor design, but more user control over the hardware resources
  ‣ Redesign the software
  ‣ Rethink the problem solving technique

• If we don’t use the parallelism, we lose it
  ‣ Amdahl’s law - serial sections
  ‣ Von Neumann bottleneck
  ‣ imbalances
CUDA Runtime

- CUDA includes a runtime library
- No explicit initialization procedure
- *Cuda C Programming Guide, §3.2*
  - Initializes the first time a runtime function is called
  - Initialized by function *other than* functions from the *device* and *version management* sections of the reference manual
  - e.g. `cudaGetLastError()`, `cudaEventCreate()`, `cudaMalloc()`
  - NOT `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`

developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/online/modules.html

- Beware of erroneous timings
- Careful when interpreting the error code from the first call into the runtime
- CUDA Context: like a CPU process (see §3.3.1)
  - Created as part of initialization
  - Host thread that initializes the run time is the only host thread with access to device memory allocated within that context, events, too (see CUDA timer in incrArr)
Asynchronous execution

- The host and device share the workload
- Some CUDA runtime calls are asynchronous
  - kernel launches
  - host ↔ device memory transfers ≤ 64KB, cudaMemcpyAsync()
- Set CUDA_LAUNCH_BLOCKING=1
  - For debugging purposes only
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
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;
    }
}
```
Device Kernel Function (cont.)

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");
```
Host code - continued

// 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 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
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>Geom</th>
<th>16 × 16</th>
<th>8 × 8</th>
<th>4 × 4</th>
<th>2×256</th>
<th>2×128</th>
<th>2×64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>1.5</td>
<td>3.2</td>
<td>4.7</td>
<td>9.8 (8.7)</td>
<td>8.5 (7.7)</td>
<td>7.4 (6.3)</td>
</tr>
<tr>
<td>Blocked</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Improved</td>
<td>125 (57)</td>
<td>53 (41)</td>
<td>12 (15)</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Using shared memory (uncoalesced glbl)

__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[tx][kk] * b[ty][kk];
            __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)

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

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

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
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 \(K \mod 16 \neq 0; \ Q \neq 1\)
• Accesses organized by half warps (16 threads) can be done in one or two transactions, under certain conditions (32, 64 and 128 byte segments)

\[
tid = blockIdx.x \times blockDim.x + threadIdx.x + K
\]
\[
\text{shared[tid]} = \text{global[tid]}
\]
\[
\text{int \ tid} = (blockIdx.x \times blockDim.x + threadIdx.x) \times Q
\]
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 $= 3*512^3/0.02097 = 18.7$ 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

© 2011 Scott B. Baden / CSE 262 / Spring 2011
Reduction
Thread Divergence

• All the threads in a warp execute the same instruction
• Different control paths are serialized
• Divergence when predicates are a function of the threadId
  
  \[
  \text{if (threadId < 2) \{ \}}
  \]
• We can avoid divergence, everyone executes the same path
  
  \[
  \text{if (threadId / WARP\_SIZE < 2) \{ \}}
  \]
• 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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..7</td>
<td></td>
<td></td>
<td></td>
<td>8..11</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8..15</td>
</tr>
</tbody>
</table>

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC
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