Lecture 17

Computing with Graphical Processing Units
Today’s lecture

• MPI wrap up
  ‣ Sorting
  ‣ Processor Geometries
• Computing with GPUs
Announcements
Bucket sort

- Divide key space into equal subranges and associate a bucket with each subrange
- Each processor maintains p local buckets
  - Assigns each key to a local bucket (local bucket \( \sim \frac{n}{p^2} \) elements)
  - Routes the buckets to the correct owner
    Sorts all incoming data into a single bucket
- Poor worst case behavior for non-uniform key distribution
Sample sort

- Overcomes poor worst case behavior
- Estimates distribution of the global key range
- Distributes key space non-uniformly over processors
- Samples keys to determine a set of $p-1$ splitters that partition the key space into $p$ disjoint intervals
- Once each processor knows the splitters, it can distribute its keys to the others accordingly
Alltoally used in sample sort


©2010 Scott B. Baden / CSE 160 / Winter 2010
Performance

• Assuming $n \geq p^3$ …
• $T_P = O\left(\frac{n}{p} \log n\right)$
• If $s = p$, each processor will merge not more than $2n/p + n/s - p$ elements
• If $s > p$, each processor will merge not more than $(3/2)(\frac{n}{p}) - (\frac{n}{ps}) + 1 + d$ elements
• Duplicates $d$ do not impact performance unless $d = O(n/p)$
• Tradeoff: increasing $s$ …
  ‣ Spreads the final distribution more evenly over the processors
  ‣ Increases the cost of determining the splitters
Performance is sensitive to processor geometry (Aliev-Panfilov on Triton.sdsc.edu)

<table>
<thead>
<tr>
<th>Geometry</th>
<th>Total GF/s</th>
<th>Total Time</th>
<th>Comm GF/s</th>
<th>Comm Time</th>
<th>Without communication GF/s</th>
<th>Without communication Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 x 4</td>
<td>38.5</td>
<td>5.02</td>
<td>0.370</td>
<td>41.6</td>
<td>4.66</td>
<td>7.2% Communication</td>
</tr>
<tr>
<td>2 x 8</td>
<td>38.4</td>
<td>5.04</td>
<td>0.410</td>
<td>41.8</td>
<td>4.63</td>
<td></td>
</tr>
<tr>
<td>8 x 2</td>
<td>38.1</td>
<td>5.07</td>
<td>0.376</td>
<td>41.2</td>
<td>4.70</td>
<td></td>
</tr>
<tr>
<td>1 x 16</td>
<td>34.6</td>
<td>5.59</td>
<td>0.965</td>
<td>41.8</td>
<td>4.62</td>
<td></td>
</tr>
<tr>
<td>16 x 1</td>
<td>33.0</td>
<td>5.87</td>
<td>1.00</td>
<td>39.8</td>
<td>4.87</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GF/s</th>
<th>Time</th>
<th>0.13</th>
<th>166.6</th>
<th>1.22</th>
<th>9.6% Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x16</td>
<td>150.4</td>
<td>1.35</td>
<td>0.13</td>
<td>166.6</td>
<td>1.22</td>
<td></td>
</tr>
<tr>
<td>16x4</td>
<td>147.3</td>
<td>1.38</td>
<td>0.14</td>
<td>164.4</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>8x8</td>
<td>146</td>
<td>1.4</td>
<td>0.17</td>
<td>165.9</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>2x32</td>
<td>133.9</td>
<td>1.52</td>
<td>0.28</td>
<td>164.6</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>1x64</td>
<td>130.9</td>
<td>1.56</td>
<td>0.32</td>
<td>164.2</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>32x2</td>
<td>126.2</td>
<td>1.61</td>
<td>0.33</td>
<td>159.2</td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>64x1</td>
<td>119</td>
<td>1.71</td>
<td>0.33</td>
<td>147.8</td>
<td>1.38</td>
<td></td>
</tr>
</tbody>
</table>

©2010 Scott B. Baden / CSE 160 / Winter 2010
Computing with Graphical Processing Units (GPUs)
Heterogeneous processing

- Two types of processors: general purpose + accelerator
  - “host” and “device”
- Accelerator can perform certain tasks more quickly subject to various overhead costs
NVIDIA GeForce GTX 280

- Hierarchically organized clusters of streaming multiprocessors
  - 240 cores @ 1.296 GHz
  - Peak performance 933.12 Gflops/s
- SIMT parallelism
- 1 GB device memory (frame buffer)
- 512 bit memory interface @ 132 GB/s

GTX 280: 1.4B transistors
Intel Penryn: 410M (dual core)
Streaming processing cluster

- GTX-280 GPU
  - 10 clusters @ 3 streaming multiprocessors or *vector cores*
- Each vector core
  - 8 scalar cores: fused multiply adder + multiplier (FMA, 32 bits), truncate intermediate result
  - May complete 1 FMA + 1 multiply per cycle = 3 flops / cycle (SP)
  - 64-bit FMA + 2 super function units (2 FMAs)
  - Share local memory (16KB) and partition the registers (64KB)
- 3 flops/core/cycle * 240 cores = 720 flops/cycle
- @ 1.296 Ghz: 933 GFLOPS
Streaming Multiprocessor
Accelerator model

- Under control of the *host*, invoke sequences of multithreaded kernels on the *device* (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions
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

©2010 Scott B. Baden / CSE 160 / Winter 2010
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
  - Grid ⊇ Block ⊇ Thread

- 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
  - Assigned to SM in units of blocks
  - Compiler re-arranges loads to hide latencies

- Global synchronization via kernel invocation
Threads, blocks, grids and data

• Threads all 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 location
  \[ \Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3 \]
  \[ \Theta(\Pi_{\iota}), \ \forall \Pi_{\iota} \in \Pi \]
Coding example – Increment Array

Serial Code

```c
void incrementArrayOnHost(float *a, int N){
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
```

```c
#include <cuda.h>
__global__ void incrementOnDevice(float *a, int N){
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+1.f;
}

incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

Rob Farber, Dr Dobb’s Journal
Managing memory

```c
float *a_h, *b_h; // pointers to host memory
float *a_d; // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i; // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);

int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
```
Transferring results + cleanup

incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
free(a_h); free(b_h);
cudaFree(a_d);
CUDA language extensions

• Type qualifiers to declare device kernel functions
  __global__ void matrixMul( …)

• Kernel launch syntax
  matrixMul<<< grid, threads >>>(…)

• Keywords
  blockIdx    threadIdx

• Runtime, e.g. storage allocation
  cudaMalloc    cudaFree    cudaMemcpy
Experiments - increment benchmark

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

N = 8388480, block size = 128, times in milliseconds

<table>
<thead>
<tr>
<th>Reps</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device time</td>
<td>8.5</td>
<td>83</td>
<td>830</td>
<td>8300</td>
</tr>
<tr>
<td>Kernel launch + data xfer</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>Sine function (Host)</td>
<td>16</td>
<td>103</td>
<td>6900</td>
<td>23.6 sec</td>
</tr>
</tbody>
</table>
Speedup

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?

• Speedup, $S$

| Running time of the fastest program on conventional processors | Running time on the accelerator |

• Baseline: a multithreaded program
Bandwidth test results

./bandwidthTest
Device 0: Tesla C1060
Host to Device Bandwidth
Transfer Size (Bytes) Bandwidth(MB/s)
32M 3324.5

Device to Host Bandwidth
32M 2792.0
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 David Kirk/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
• Shared memory and registers do not persist across kernel invocations
• 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
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
  ‣ Global memory accesses → fast on-chip accesses
  ‣ Coalesced memory transfers
• Avoid costly branches, or render them harmless
Is it that simple?

- Simplified processor design, but more user control over the hardware resources
  - Rethink problem solving technique & implementation
  - Redesign the software

- If we don’t use the parallelism, we lose it
  - Amdahl’s law - serial sections
  - Von Neumann bottleneck – data transfer costs
  - Workload Imbalances