Today’s lecture

• Short topic
  ‣ Aliev-Panfilov on 64 cores of Triton

• GPUs
  ‣ Under the hood of the device
  ‣ Applications
  ‣ Managing locality
Announcements
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>
<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>9.6% Communication</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>146.1</td>
<td>1.40</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>119.1</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 260/ Fall '10
Under the hood of the device
Streaming Multiprocessor
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 260/ Fall '10
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
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
- Bound to and only accessible from their thread until the block finishes execution
Warp scheduling

- Blocks are divided into warps of 32 (SIMD) threads which are ….
- Subdivided into schedulable units: 16 threads (→ 32 on Fermi)
- Zero scheduling overhead - in hardware
- Scheduler finds an eligible warp: all operands are ready
- Branches serialize execution in a warp
- 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]
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

- Blocks
  - ≤ 512 threads
  - Registers subdivided over threads

- Synchronization among all threads in the block, only within block
Occupancy

- Ratio: \# active warps ÷ max \# warps supported by vector unit
- Limited by vector units resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of thread
- Minimum \# active warps required to hide latency
- 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
“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
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
Aliev-Panfilov implementation under CUDA

\[ E_t[i,j] = E_{t-1}[i,j] + \alpha(E_{t-1}[i+1,j] + E_{t-1}[i-1,j] + E_{t-1}[i,j+1] + E_{t-1}[i,j-1] - 4E_{t-1}[i,j]) \]
Naïve Aliev-Panfilov Code

• All array references go through device memory
• ./apf -n 6144 -t 0.04, 16x16 thread blocks
• Single Prec: 31 Gflops [Triton, 32 cores, MPI: 85GF]
• Double Prec: 17 Gflops (19GF n=8192) [Triton: 48 GF]

#define E′ [i,j] E_prev[(j+1)*(m+3) + (i+1)]
I = blockIdx.y*blockDim.y + threadIdx.y;
J = blockIdx.x*blockDim.x + threadIdx.x;
if ((I <= n) && (J <= m) )

for (j=1; j<= m+1; j++)
    for (i=1; i<= n+1; i++)
        E[j][i] = E′ [j][i]+α*(E′ [j][i-1]+E′ [j][i+1] +
                         E′ [j-1][i]+E′ [j+1][i] - 4*E′ [j][i]);
Optimized Aliev-Panfilov Code

- Use shared memory:
  31 Gflops → 126 Gflops
  85 Gflops on 32 Triton cores with MPI

```c
__shared__ float block[BDIM_Y + 2][BDIM_X + 2];
int idx = threadIdx.x, idy = threadIdx.y;
int y = blockIdx.y * (BDIM_Y) + idy;
int x = blockIdx.x * (BDIM_X) + idx; idx++; idy++;
unsigned int index = (y+1) * (m+3) + (x+1);

block[idy][idx] = center = E_prev[index];
__syncthreads();
e = center + α*(block[idy][idx-1] + block[idy][idx+1] +
                   block[idy-1][idx] + block[idy+1][idx] - 4*center);
e = e - dt*(kk * e * ( e - a) * ( e - 1 ) + e * r); 
E[index] = e;
```
Ghost Cells

• Introduce branches and thread divergence

    int idx = threadIdx.x, idy = threadIdx.y;
    int y = blockIdx.y * (BDIM_Y) + idy++;
    int x = blockIdx.x * (BDIM_X) + idx++;
    unsigned int index = y * N + x
    if(idy == 1 && y > 0 ){
        //read a ghost cell into shared memory
        block[0][idx] = E_prev [index-N];

    Ghost cells

    When loading ghost cells, only some of the threads are active, some are idle
Ghost Cells (cont.)

• Divide the work between threads so that
  ‣ each thread is responsible for 1 ghost cell load

• For a tile size of 16 x 16
  ‣ There are 16x4 = 64 ghost cells
  ‣ Then, create 64 threads
Synchronizzazione

- void __syncthreads()
- Synchronizes all threads in a block
- Threads wait until all have arrived
- Avoids memory hazards
- Beware of conditionals
  if (condition(x)) __syncthreads();
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
  
  ```
  if (threadId < 2) {} 
  ```
- We can avoid divergence, everyone executes 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
</tr>
<tr>
<td>2</td>
<td>0...3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0..7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

©2010 Scott B. Baden /CSE 260/ Fall '10
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]);
}
```

©2010 Scott B. Baden /CSE 260/ Fall '10
Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?

Ruye Wang,
fourier.eng.hmc.edu/e85/lectures/memory
Bank conflicts

• Access to shared memory “as fast” as registers unless...
• 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;
a[idx] = a[idx] + 1.0f;
Memory coalescing

• Memory banks: consecutive addresses can be read very quickly (K=0; Q=1)
• Certain non-sequential access patterns to global memory degrade performance K mod 16 ≠ 0; Q≠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)

\[
tid = blockIdx.x*blockDim.x+threadIdx.x + K
\]
\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}]
\]
\[
\text{int } tid = (\text{blockIdx.x*blockDim.x + threadIdx.x})*Q
\]
Memory coalescing

- Simplest: addresses are contiguous across threads
- Accesses organized by half warps (16 threads)

```c
__shared__ a[ ][ ], b[ ][ ];
I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;
a[ty][tx] = A[I*N+k*by+tx];
b[ty][tx] = B[J+N*(k*bx+ty)];
```

Threads →  
Tile iteration

Nvidia Corp.
Bank conflicts in 2d arrays

- All warps in a block access consecutive elements within a row as they step through neighboring columns (matrix multiply)

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

- Accesses by threads in a block along a column don’t coalesce
Reducing divergence and avoiding bank conflicts

Thread 0

©2010 Scott B. Baden /CSE 260/ Fall '10
The Code

```cpp
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];
}
```

©2010 Scott B. Baden /CSE 260/ Fall '10
Fin