Taking CUDA to Ludicrous Speed

Getting Righteous Performance from your GPU

Blue Waters Undergraduate Petascale Education Program
May 29 – June 10 2011
LIGHTSPEED IS TOO SLOW. WE’LL HAVE TO GO RIGHT TO LUDICROUS SPEED.
Optimizing on CPUs

- Could I be getting better performance?
  - Probably a little bit, but most of it is hidden from the user.
- How much better?
  - If you compile --O3, you can get faster (maybe 2x)
  - If you are careful about tiling your memory, you can get faster on codes that benefit from that (maybe 2-3x)
- Is that much performance worth the work?
  - Compiling with optimizations is a no-brainer (and yet…)
  - Tiling is useful, but takes an investment
Optimizing on GPUs

- “But my code already runs fast in CUDA”
- How much faster is possible?
  - Near peak GFLOP performance?
- How much work would you do for a 17x speedup?
  - How about 400x?
- Most of the modifications are straightforward.
  - You just need to know the hardware.

---

Particle Code
1 time-step, n=4096)

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Naïve GPU</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>653</td>
<td>26.6</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Speedups:
- 24x
- 17x
Sidebar: Timers

- CUDA Event Timers
  - Gives highly accurate times (~ nanosecond precision)

```c
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );
kernel<<<grid,threads>>>( d_odata, d_idata, size_x, size_y, NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
cudaEventElapsedTime( &time, start, stop );  //gives time in milliseconds
cudaEventDestroy( start );
cudaEventDestroy( stop );
```
3 Easy* Steps to Better Performance

- Or: “How not to shoot yourself in the foot”

1. Memory Hierarchy
2. Execution Configuration
3. Instruction/Flow Optimization

* Okay, so not that easy...
Memory Hierarchy

- Global
  - Everyone can access it
  - cudaMemcpy goes into here
  - SLOW!! (off-chip)

- Shared
  - Visible to threads in the same block
  - ~100x faster than global

- Register
Memory Hierarchy

Don’t Shoot Yourself in the Foot—Rule #1: Know where your data is!

<table>
<thead>
<tr>
<th>Memory</th>
<th>Scope</th>
<th>Relative Speed (in number of instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Thread</td>
<td>1</td>
</tr>
<tr>
<td>Local</td>
<td>Thread</td>
<td>200+</td>
</tr>
<tr>
<td>Shared</td>
<td>Block</td>
<td>~2-3</td>
</tr>
<tr>
<td>Global</td>
<td>All</td>
<td>200+</td>
</tr>
<tr>
<td>Constant</td>
<td>All</td>
<td>200+</td>
</tr>
<tr>
<td>Texture</td>
<td>All</td>
<td>2 – 200 (cached)</td>
</tr>
</tbody>
</table>
Don’t Shoot Yourself in the Foot—Rule #1: Know where your data is!

<table>
<thead>
<tr>
<th>Memory</th>
<th>Scope</th>
<th>Relative Speed (in number of instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Thread</td>
<td>1</td>
</tr>
<tr>
<td>Local</td>
<td>Thread</td>
<td>200+</td>
</tr>
<tr>
<td>Shared</td>
<td>Block</td>
<td>~2-3</td>
</tr>
<tr>
<td>Global</td>
<td>All</td>
<td>200+</td>
</tr>
<tr>
<td>Constant</td>
<td>All</td>
<td>200+</td>
</tr>
<tr>
<td>Texture</td>
<td>All</td>
<td>2 – 200 (cached)</td>
</tr>
</tbody>
</table>
Coalesce Global Memory Accesses

- Global memory is slow but unavoidable
- Accesses should be *coalesced*
  - Accesses from a single warp can be grouped into a single load.
  - Keep accesses sequential and close to each other

![Diagram showing global memory access patterns with coalesced and non-coalesced access examples.](image-url)
Shared Memory

- Limited space is available
  - Tesla: 16 kB per Multiprocessor, Fermi: 64 kB
- Therefore, load a *tile* from global memory, use this data, then load the next tile
- `__shared__` declares a variable in shared memory
- Each thread loads a small portion of the data
- Then all threads in a block can get it from shared memory much faster
Synchronization

- Trade-offs to having so many cores
  - All have to run the same kernel function
  - Don’t know what order blocks will be executed
- Luckily we do have one way to coordinate threads:
  - `__syncthreads()`
    - Barrier: threads in a block wait until everyone gets there
    - Be careful putting these inside ‘if’ blocks
    - Use these to make sure all threads have loaded their data before trying to use it
Taking CUDA to Ludicrous Speed

Tiling in Shared Memory

```c
const int threads_per_block = 256;

__global__ void add_array(double * dev_array) {
    __shared__ tile[threads_per_block];

    int i = blockIdx.x * blockDim.x + threadIdx.x;

    //load tile into shared memory
    tile[threadIdx.x] = dev_array[i];

    __syncthreads(); //wait for all threads to get here

    double total;
    for (int j=0; j < threads_per_block; j++) {
        total += tile[threadIdx.x] * tile[j];
    }
    dev_array[i] = total;
}
```
Don’t Shoot Yourself in the Foot—
Rule #2: Keep your GPU occupied.

dim3 grid(3, 2);
dim3 block(4, 3);

kernel <<< grid, block >>> (a, b);
Mapping to Hardware

- **Grids** map to GPUs
- **Blocks** map to the MultiProcessors (MP)
  - Blocks are never split across MPs, but an MP can have multiple blocks.
  - The more blocks it has available to schedule, the more work it can get done.
Mapping to Hardware

- **Threads** map to Stream Processors (SP)
- **Warps** are groups of (32) threads that execute simultaneously
  - Controlled by “Instruction Unit”
Maximizing Execution

- Number of threads per block matters!
- Each block lives on a single Multiprocessor
- Therefore:
  - More total threads allow you to hide memory latency
  - Better use of shared memory
  - More likely to be using all your cores
- But:
  - Make sure each thread has enough to do
  - Limited amount of threads per block, blocks per grid, registers per block, shared memory per block…
Execution Configuration

- Design your algorithm so that each thread has its own piece of work to do
  - e.g. calculate forces for a single particle
  - calculate a number of intervals for area under curve

- Try different numbers of threads per block, timing each one to see which configuration is fastest
  - Use CUDA_Occupancy_calculator.xls to make more intelligent guesses
Optimization Tools

- CUDA_Occupancy_calculator.xls

- CUDA Profiler
Instruction Optimization

Don’t Shoot Yourself in the Foot—
Rule #3: If all else fails, just do less work.

- Only try out instruction-level optimizations after you’ve maximized memory and occupancy
- Some techniques:
  - Use `float` where double precision isn’t necessary
  - Minimize divergent branches
  - Unroll loops
- More can be found in CUDA documentation
Ludicrous Speed

- Remember to verify your results every step of the way: No point in getting the wrong answer really really fast!
- Optimize memory accesses first!
  - It’s the simplest thing to fix and makes the largest impact
- Don’t forget about the overhead to copying data back and forth: do as much work on the GPU as possible