Massively Parallel Processing on the GPU

Mentees:
Eric Joyner
Ian Wetherbee

Mentor: John Stratton
Why Parallel?

- Current chips have hit a speed ceiling
- Performance only possible by taking advantage of increasing # of cores
CUDA and OpenCL

• We did most of our programming in CUDA
  - “Compute Unified Device Architecture”
  - Created by Nvidia
  - Extension to C

• GPU is massively parallel
  - Single Program, Multiple Data
  - Many lightweight threads running concurrently

Challenges

- Writing a program for a GPU requires thinking about execution differently from a CPU
  - Find concurrency
  - Probably need to split program into blocks, threads
  - See: matrix multiplication, vector (array) scan as examples
Kernel

- Kernel program is a grid split into blocks, threads
  - Depending on GPU version, there can be 512 or 1024 threads in block
  - 3D thread blocks
  - 2D grid
  - Any number of blocks
Bottlenecks

- Each kernel must run thousands of threads to compensate for:
  - Memory transfer time Host ↔ GPU
  - Parallel memory architecture
    - Most GPUs handle memory access in bundles of 16 words, but only to the same or successive addresses
  - Access times to global vs shared memory
GPU code structure

Function calling GPU kernel:
1. Allocate memory on the device
2. Copy input data to the device
3. Run the device kernel
4. Copy output from device to host code
5. Free GPU memory
Example: Sum of Differences (1)

```c
__kernel void templmatch_per_result(__global uchar* search, ...) {
  ulong r = 0;
  uint absd, start, tx, ty;
  tx = get_global_id(0);
  ty = get_global_id(1);
  start = mad24(ty,search_step,tx);

  for (uint j=0; j<templ_height; j++) {
    for (uint i=0; i<templ_width; i++) {
      absd = abs_diff(search[start + j*search_step+i],
                       templ[j*templ_step+i]);
      r += absd*absd;
    }
  }
  result[ty*result_width+tx] = r;
}
```

Outer 2 `for` loops are launched as rows*cols threads in a kernel

Inner loop uses built-in functions to figure out what thread it is → what location to calculate

for (int i=0; i < result.rows; i++) {
  for (int j=0; j < result.cols; j++) {
    // sum for this template at offset i,j
    for (int it=0; it < templ.rows; it++) {
      for (int jt=0; jt < templ.cols; jt++) {
        result.data[i*result.step+j] +=
        pow(templ.data[it*templ.step+jt] -
        search.data[(i+it)*search.step+j+jt], 2);
      }
    }
  }
}
__kernel void templmatch_columns(__global uchar* search, ..., uchar reduction_size) {

__local ulong cols[256];
uint gx, gy, tx, start, absd;
gx = get_group_id(0); // result location
gy = get_group_id(1);
tx = get_local_id(0); // column
start = mad24(gy,search_width,gx);

for (uint row=0; row < templ_height; row++) {
    absd = abs_diff(search[start + mad24(row,search_step,tx)],
                    templ[mad24(row,templ_step,tx)]);
    cols[tx] += absd*absd;
}
barrier(CLK_LOCAL_MEM_FENCE);

for (uint c = reduction_size >> 1; c > 0; c >>= 1) {
    if (tx < c && tx < templ_width) { // only reduce valid columns
        cols[tx] = cols[tx] + cols[tx+c];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
result[gy*result_width + gx] = cols[0];
}
Execution Time (seconds)

- CPU Unoptimized: 88 seconds
- GPU Unoptimized: 0.14 seconds
- GPU Optimized: 0 seconds
Summary

- Parallel programming requires a different mindset
  - Avoid bottlenecks
  - Create many threads
- Dramatic speedups if optimized properly
  - Difference between ~10x to ~100x
Questions