Let's Look at Optimizing an OpenCL Program

- Simple PDE solver (2x 4096x4096 grid of floats; 64MB of data)
  - 5-point stencil computation for all points
  - Convergence estimation (max-min)
  - Repeat until converged

Test System:
- Sandy Bridge i7-2600
- 3.4GHz (4-core)
- AMD Radeon HD 6950M

### C Code

```c
int main (int argc, const char * argv[]) {
    float range = BIG_RANGE;
    float *in, *out;
    // Initialize create_data(&in, &out);
    // Compute Range range = find_range(&in, &out);
    // Calculate update_ranges(range, out, SIZE);
}
```

### Profiling the C Code

1. Baseline

![Graph showing performance data]

### Replacing update() with OpenCL

```c
int main (int argc, const char * argv[]) {
    float range = BIG_RANGE;
    float *in, *out;
    // Initialize create_data(&in, &out);
    // Compute Range range = find_range(&in, &out);
    // Update OpenCL
    update_cl();
}
```

### update_cl()

```c
void update_cl(float *in, float *out) {
    cl_int error;
    // Load the program source
    char *program_text = load_source_file("kernel.cl");
    // Create the program
    cl_program program = createProgramWithSource(opencl_context, 1, (const char**) &program_text, NULL, &error);
    // Compile the program and check for errors
    error = clBuildProgram(program, 1, &opencl_device, NULL, NULL, NULL);
    // Create the computation kernel
    cl_kernel kernel = clCreateKernel(program, "update", &error);
}
```
update_cl()

// Create the data objects
in_buffer = clCreateBuffer(opencl_context, CL_MEM_READ_ONLY, SIZE_BYTES, NULL, &error);
in_buffer = clCreateBuffer(opencl_context, CL_MEM_WRITE_ONLY, SIZE_BYTES, NULL, &error);

// Copy data to the device
error = clEnqueueWriteBuffer(opencl_queue, in_buffer, CL_FALSE, 0, SIZE_BYTES, out, 0, NULL, &error);

// Set the kernel arguments
element = clSetKernelArg(kernel, 0, sizeof(in_buffer), &in_buffer);
element = clSetKernelArg(kernel, 1, sizeof(out_buffer), &out_buffer);

// Enqueue the kernel
element, out, NULL, NULL, NULL);

clEnqueueNDRangeKernel(opencl_queue, kernel, 0, NULL, &error);

void Main()

int WIDTH = get_global_size(0);
int HEIGHT = get_global_size(1);

if (get_global_id(0) == 0 || get_global_id(1) == 0) return;

int x = get_global_id(0);
int y = get_global_id(1);

float a = in_buffer[(x * WIDTH) + y];
float b = in_buffer[(x * WIDTH) + y + 1];
float c = in_buffer[(x * WIDTH) + y + 2];
float d = in_buffer[(x * WIDTH) + y + 3];

// Do the computation and write back the result
out_buffer[(x * HEIGHT) + y] = (a + b) * (c + d);

return;

Profiling OpenCL

More Detailed Profiling

Start_perf_measurement(Read_perf);

// Enqueue a read to get the data back
error = clEnqueueReadBuffer(opencl_queue, out_buffer, CL_FALSE, 0, SIZE_BYTES, out, 0, NULL, &error);

// Wait for it to finish
error = clFinish(opencl_queue);
checkError(error, "clFinish(opencl_queue)");

Start_perf_measurement(finish_perf);

// Wait for it to finish
Start_perf_measurement(finish_perf);
checkError(error, "clFinish(opencl_queue)");

Stop_perf_measurement(finish_perf);

Stop_perf_measurement(Read_perf);

Stop_perf_measurement(Read_perf);

This is pretty stupid.
Why are we compiling the kernel every time we call update_cl()??
More Detailed Profiling Information

2. Baseline with Profiling

What’s Going On?

• All of our time in clFinish()
• None of our time in update()

• OpenCL is asynchronous. You enqueue actions, but the call returns immediately.
• clFinish() causes OpenCL to wait until everything is done.
  – So our code takes zero time until we get to clFinish() and then all of our time is counted at once!
• Solution: insert clFinish() everywhere we want to force it to finish to time the code.
  – Note: you can use event info to get this without forcing a finish, which is faster if you have a lot going on, but doesn’t matter here.

Doing Profiling That Works

start_perf_measurement(read_perf);
// Enqueue a read to get the data back
error = clEnqueueReadBuffer(openL_queue, out_buffer, CL_FALSE, 0, SIZE*SIZE, NULL, openL_queue);
clFinish(openL_queue);
// Wait for it to finish
start_perf_measurement(clFinish_perf);
error = clFinish(openL_queue);
checkError(error, “clFinish”);
stop_perf_measurement(clFinish_perf);

Move the Overhead Out of the Loop

• We only need to do these once:
  – Compile program
  – Setup the buffers
  – Copy the data to the device
• We still need to do these every time:
  – Read back the results (for range())
  – Enqueue the kernel (to do the calculation)
  – Wait for it to finish

More Detailed Profiling Information

3. Baseline with Profiling (and clFinish)

New Main Loop

// ======== Compute Iteration
while (range > LIMIT) {
  // Calculation
  start_perf_measurement(update_perf);
  update_clipter_k4_buffers(), get_out_buffer());
  stop_perf_measurement(update_perf);
  // Read back the data
  start_perf_measurement(read_perf);
  read_back_data(get_out_buffer(), out);
  stop_perf_measurement(read_perf);
  // Compute Range
  start_perf_measurement(range_perf);
  range = find_range(out, SIZE*SIZE);
  stop_perf_measurement(range_perf);
  iterations++;
  printf(“Iteration %d, range=\%d
”, iterations, range);
}
New update_cl() and read_back_data()

```c
void update_cl(cl_mem in_b, cl_mem out_b) {
    cl_int error;
    // Set the kernel arguments
    ...checkError...
    error = clEnqueueNDRangeKernel(queue, kernel, 3, 0, NULL, &num);  // Ignore the border error
    ...checkError...
    }
    
    void read_back_data(cl_mem buffer_to_read_from, float *result_buffer) {
        cl_int error;
        // Enqueue a read to get the data back
        ...checkError...
        clFinish(queue);
    }
```

Performance With Reduced Overhead

![Chart](image)

4. Overhead Outside of Loop

Analysis

- Update is now very fast on the GPU and 4x faster on the 4-core CPU. (Good work!)
- However...
  - GPU:
    - 34% reading (transferring the data)
    - 46% update (on the CPU)
  - CPU:
    - 16% reading (transferring the data??)
    - 48% update

(We could use OpenCL's map and upmap functions to map the data into the application’s space and thereby avoid this overhead on the CPU)

Next Step

- To eliminate the time reading the data we need to keep the data on the device (GPU)
- To do this we need to move the range() function to the GPU.
- But the range() is a reduction, so we need synchronization across all threads on the device...

Putting range() on the Device

```c
// Compute while range > 20k
{
    // Calculation
    ...start_perf_measurement...
    ...stop_perf_measurement...
    // Range
    ...start_perf_measurement...
    ...stop_perf_measurement...
    // Read back the data
    ...start_perf_measurement...
    ...stop_perf_measurement...
    // Compute Range
    ...range
    printf(...iteration %d, range_min, range_max);
    ...}
```

range() kernel

- The range() function is a reduction. That means we need synchronization across the whole kernel.
- So we do this in two steps:
  - 1) Divide the data into 4096 chunks and calculate min/maxes for each in parallel on the device
  - 2) Read back the 4096 min/max values and calculate the final min/max on the CPU
- This reduces the data transfer from the full data set to just 4096 values.

```
Out:    Range:
0-15    16-31    32-47
16-31    32-47    48-63
...      ...
```

02/12/2011
range() kernel
kernel void range(global float *data, int total_size, global float *range) {
float min, max;
// Find out which items this work-item processes
int size_per_workitem = total_size / get_num_groups(0);
int start = size_per_workitem * get_global_id(0);
// Find the min/max for our chunk of the data
min = data[start];
for (int i = start + size_per_workitem; i < start + size_per_workitem + size_per_workitem; i++) {
if (data[i] < min) min = data[i];
else if (data[i] > max) max = data[i];
}
// Write the min and max back to the range we will return to the host
range[get_global_id(0)] = min;
range[get_global_id(0)+1] = max;
}  
• Partial reduction: each kernel gets a chunk of the data and reduces it to a min/max value.

Putting range() on the Device

What’s Going On?
• Read data: only 65% faster, but we reduced the data from 64MB to 32kB.
  – Per-transfer overhead (setup/synchronization)
  – Not just data transfer time
• Range kernel: 3x slower.
  – What’s going on?

Analyzing Memory Accesses
Assume DRAM Width = 128 bytes = 4 floats
Uncoalesced

Fixing the range() kernel
kernel void range_coalesced(global float *data, int total_size, global float *range) {
float min, max;
// Work-items in a work-group process neighboring elements on each iteration
int size_per_workgroup = total_size / get_num_groups(0);
int start = size_per_workgroup * get_global_id(0) + get_local_id(0);
// Each work-item finds the min/max for its chunk of the data
min = max = 0.0f;
for (int i = start; i < start + size_per_workgroup; i++) {
if (data[i] < min) min = data[i];
else if (data[i] > max) max = data[i];
}
// Write the min and max back to the range we will return to the host
range[get_global_id(0)] = min;
range[get_global_id(0)+1] = max;
}
Coalesced

Coalesced range() Performance

CL-CPU

02/12/2011
Conclusions

- Measuring performance for asynchronous execution is tricky
  - You should really use tools to do this, but hard to automate
- Moving data is expensive
  - Per transfer overhead
  - Limited bandwidth
- Memory access optimizations are essential
- Limited synchronization can be difficult
Questions?