UPMARC Uppsala Programming for Multicore Architectures Research Center

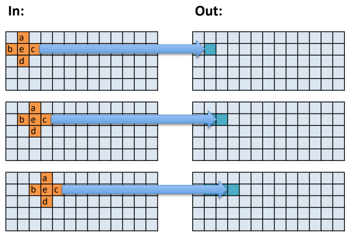
Accelerating Code with OpenCL

David Black-Schaffer
Assistant Professor, Department of Information Technology
Uppsala University

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 6980M



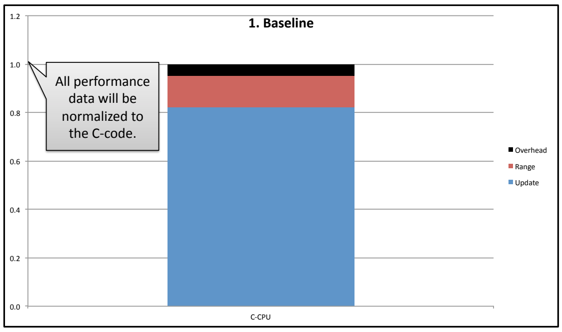
C Code

```
int main (int argc, const char * argv[]) {
    float range = BIG_RANGE;
    float *in, *out;

    // ======== Initialize
    create_data(&in, &out);

    // ======== Compute
    while (range > LIMIT) {
        // Calculation
        update(in, out);

        // Compute Range
        range = find_range(out, SIZE*SI

        swap(&in, &out);
    }
}
```

```
void update(float *in, float *out) {
    for (int y=1; y<SIZE-1; y++) {
     for (int x=1; x<SIZE-1; x++) {
          float a = in[SIZE*(y-1)+(x)];
          float b = in[SIZE*(y)+(x-1)];
          float c = in[SIZE*(y+1)+(x)];
          float d = in[SIZE*(y)+(x+1)];
          float e = in[SIZE*y+x];
          out[SIZE*y+x] = (0.1*a+0.2*b+0.2*c+0.1*d+0.4*e);
     }
    }
}
```

```
float find_range(float *data, int size) {
    float max, min;
    max = min = 0.0f;
    // Iterate over the data and find the min/max
    for (int i=0; i<size; i++) {
     if (data[i] < min)
          min = data[i];
     else if (data[i] > max)
          max = data[i];
    }
    // Report the range
    return (max-min);
}
```

Profiling the C Code



Replacing update() with OpenCL

```
int main (int argc, const char * argv[]) {
   float range = BIG_RANGE;
   float *in, *out;

   // ======== Initialize
   init_all_perfs();
   create_data(&in, &out);

   // ======== Setup OpenCL
   setup_cl(argc, argv, &opencl_device, &opencl_context, &opencl_queue);

   // ======== Compute
   while (range > LIMIT) {

    // Calculation
    update_cl(in, out);

    // Compute Range
    range = find_range(out, SIZE*SIZE);

    iterations++;
    swap(&in, &out);

    printf("Iteration %d, range=%f.\n", iterations, range);
   }
}
```

Boring code to find devices and initialize queues.

This is were the work happens.

---

update_cl()

```
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;
   program = clCreateProgramWithSource(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()

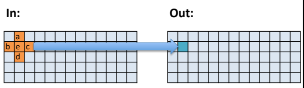
```
   // Create the data objects
   cl_mem in_buffer, out_buffer;
   in_buffer = clCreateBuffer(opencl_context, CL_MEM_READ_ONLY,
SIZE_BYTES, NULL, &error);
   out_buffer = clCreateBuffer(opencl_context, CL_MEM_WRITE_ONLY,
SIZE_BYTES, NULL, &error);

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

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

   // Enqueue the kernel
   size_t global_dimensions[] = {SIZE,SIZE,0};
   error = clEnqueueNDRangeKernel(opencl_queue, kernel, 2, NULL,
global_dimensions, NULL, 0, NULL, NULL);
```


update_cl()

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

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

   // Cleanup
   clReleaseMemObject(out_buffer);
   clReleaseMemObject(in_buffer);
   clReleaseKernel(kernel);
   clReleaseProgram(program);
   free(program_text);
}
```


Kernel Code

```
kernel void update(global float *in, global float *out) {
   int WIDTH = get_global_size(0);
   int HEIGHT = get_global_size(1);

   // Don't do anything if we are on the edge.
   if (get_global_id(0) == 0 || get_global_id(1) == 0)
    return;
   if (get_global_id(0) == (WIDTH-1) || get_global_id(1) == (HEIGHT-1))
    return;
   int y = get_global_id(1);
   int x = get_global_id(0);

   // Load the data
   float a = in[WIDTH*(y-1)+(x)];
   float b = in[WIDTH*(y)+(x-1)];
   float c = in[WIDTH*(y+1)+(x)];
   float d = in[WIDTH*(y)+(x+1)];
   float e = in[WIDTH*y+x];
   // Do the computation and write back the results
   out[WIDTH*y+x] = (0.1*a+0.2*b+0.2*c+0.1*d+0.4*e);
}
```

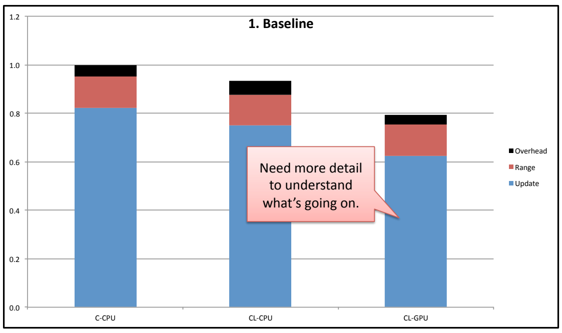


update_cl()

1. Load the code for the kernel from a text file
2. Create a CL program from the kernel code
3. Build (compile) the program
4. Create a kernel to call
5. Create the buffers for the data
6. Write the data to the buffers
7. Set the arguments for the kernel
8. Enqueue the kernel
9. Read the results back
10. Wait for OpenCL to finish
11. Cleanup

This is pretty stupid.
Why are we compiling the kernel **every** time we call update_cl()?

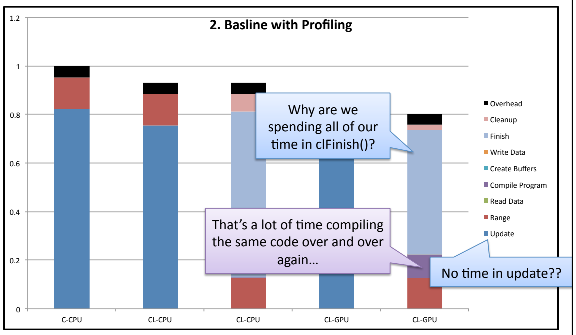
Profiling OpenCL



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_B
checkError(error, "clEnqueueReadBuffer");
stop_perf_measurement(&read_perf);

// Wait for it to finish
start_perf_measurement(&finish_perf);
error = clFinish(opencl_queue);
checkError(error, "clFinish");
stop_perf_measurement(&finish_perf);
```

More Detailed Profiling Information



What's Going On?

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

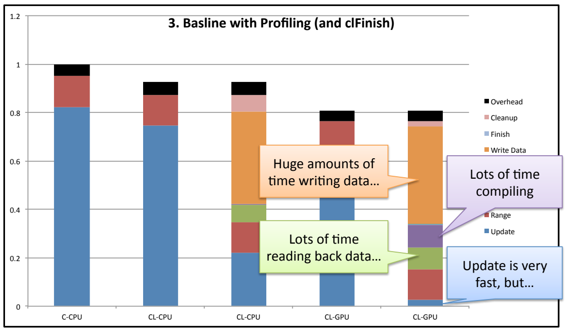
- OpenCL is **asynchronous**. You **enqueue** actions, but the call returns immediately.
- clFinish() causes OpenCL to wait until everything is done.
  - So our code takes zero time until we get to clFinish() and then all of our time is counted at once!
- Solution: insert clFinish() everywhere we want 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(opencl_queue, out_buffer, CL_FALSE, 0, SIZE_B
checkError(error, "clEnqueueReadBuffer");
clFinish(opencl_queue);
stop_perf_measurement(&read_perf);

// Wait for it to finish
start_perf_measurement(&finish_perf);
error = clFinish(opencl_queue);
checkError(error, "clFinish");
stop_perf_measurement(&finish_perf);
```

Forces the clEnqueueReadBuffer() to finish before the code continues. Bad for performance, but good for measurements.

More Detailed Profiling Information



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

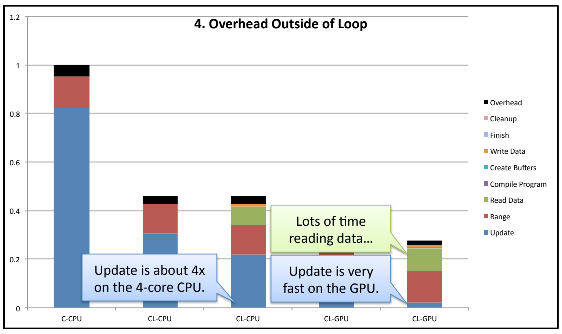
New Main Loop

```
// ======== Compute
while (range > LIMIT) {

    // Calculation
    start_perf_measurement(&update_perf);
    update_cl(get_in_buffer(), 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=%f.\n", iterations, range);
}
```

New update_cl() and read_back_data()

```
void update_cl(cl_mem in_b, cl_mem out_b) {
    cl_int error;
    // Set the kernel arguments
    error = clSetKernelArg(update_kernel, 0, sizeof(in_b), &in_b);
    checkError(error, "clSetKernelArg in");
    error = clSetKernelArg(update_kernel, 1, sizeof(out_b), &out_b);
    checkError(error, "clSetKernelArg out");

    // Enqueue the kernel
    size_t global_dimensions[3] = {SIZE,SIZE,0}; // Ignore the border
    error = clEnqueueNDRangeKernel(opencl_queue, update_kernel, 2, NULL, g
    checkError(error, "clEnqueueNDRangeKernel");
    clFinish(opencl_queue);
}


void read_back_data(cl_mem buffer_to_read_from, float *result_buffer) {
    cl_int error;
    // Enqueue a read to get the data back
    error = clEnqueueReadBuffer(opencl_queue, buffer_to_read_from, CL_FALSE,
    checkError(error, "clEnqueueReadBuffer");
    clFinish(opencl_queue);
}
```


Performance With Reduced Overhead



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

Why are we spending time transferring the data for the OpenCL CPU version?

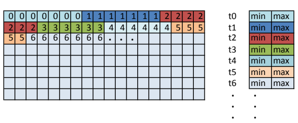
(We could use OpenCL's map and upmap functions to map the data into the application's space and thereby avoid this overhead on the CPU.)

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…

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.



Putting range() on the Device

```
// ======== Compute
while (range > LIMIT) {

    // Calculation
    start_perf_measurement(&update_perf);
    update_cl(get_in_buffer(), get_out_buffer());
    stop_perf_measurement(&update_perf);

    // Range
    start_perf_measurement(&range_perf);
    range_cl(get_out_buffer());
    stop_perf_measurement(&range_perf);

    // Read back the data
    start_perf_measurement(&read_perf);
    read_back_data(range_buffer, range_data);
    stop_perf_measurement(&read_perf);

    // Compute Range
    start_perf_measurement(&reduction_perf);
    range = find_range(range_data, RANGE_SIZE*2);
    stop_perf_measurement(&reduction_perf);

    iterations++;

    printf("Iteration %d, range=%f.\n", iterations, range);
```

Calculate the min/max for 4096 chunks of the data in parallel.

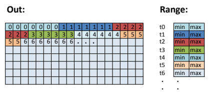
Read back the 4096 min/max values. (32kB instead of 64MB.)

On the CPU do the final min/max reduction over the 4096 min/ max values.

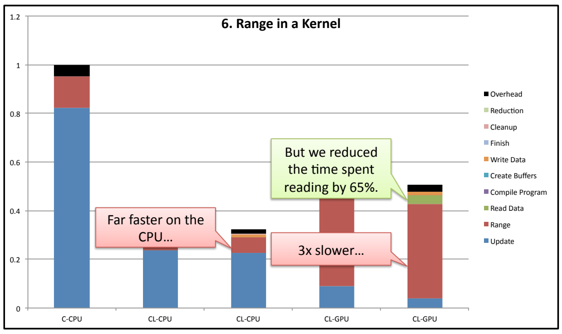
range() kernel

```
kernel void range(global float *data, int total_size, global float *range) {
    float max, min;

    // Find out which items this work-item processes
    int size_per_workitem = total_size/get_global_size(0);
    int start = size_per_workitem*get_global_id(0);
    int stop = start+size_per_workitem;

    // Finds the min/max for our chunk of the data
    min = max = 0.0f;
    for (int i=start; i<stop; 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)*2] = min;
    range[get_global_id(0)*2+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



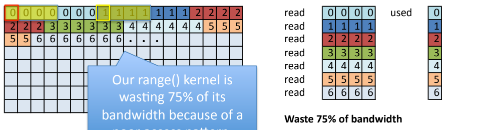
What's Going On?

- 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?

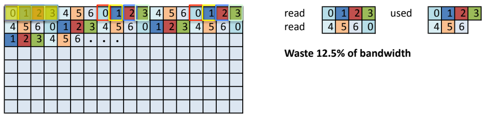
Analyzing Memory Accesses

Assume DRAM Width = 128 bytes = 4 floats

Uncoalesced



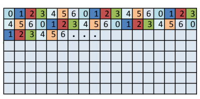
Coalesced



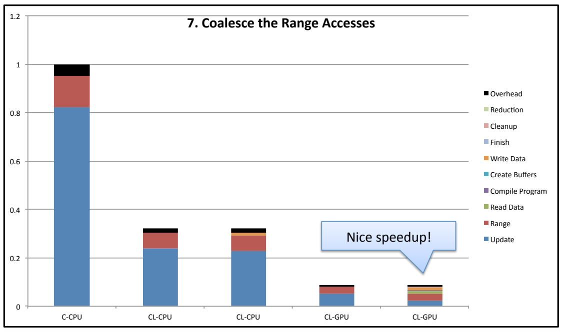
Fixing the range() kernel

```
kernel void range_coalesced(global float *data, int total_size, global float
*range) {
    float max, min;

    // 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_group_id(0)+get_local_id(0);
    int stop = start+size_per_workgroup;

    // Each work-item finds the min/max for its chunk of the data
    min = max = 0.0f;
    for (int i=start; i<stop; i+=get_local_size(0)) {
        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
    range[get_global_id(0)*2] = min;
    range[get_global_id(0)*2+1] = max;
}
```

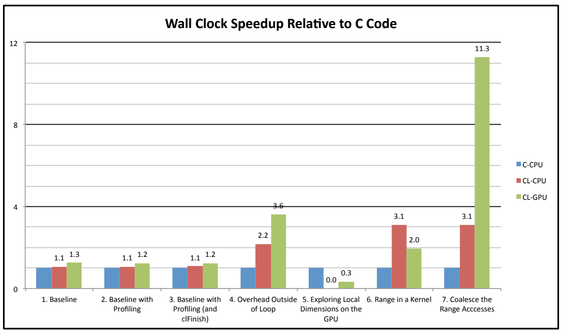
Coalesced



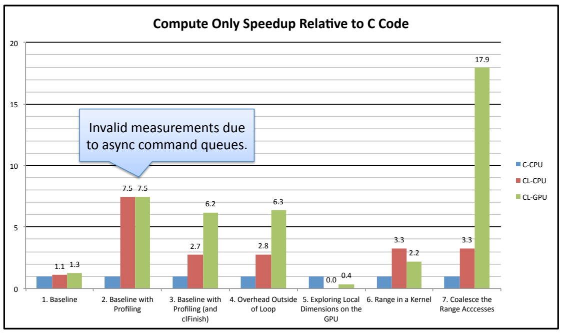
Coalesced range() Performance



Overall Speedups

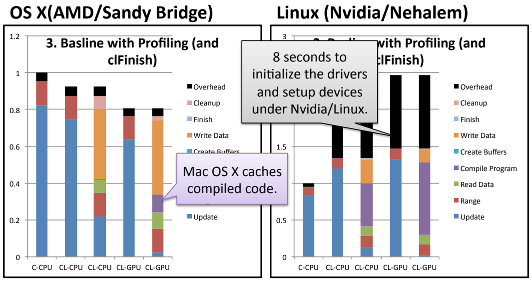


Just the Compute Part



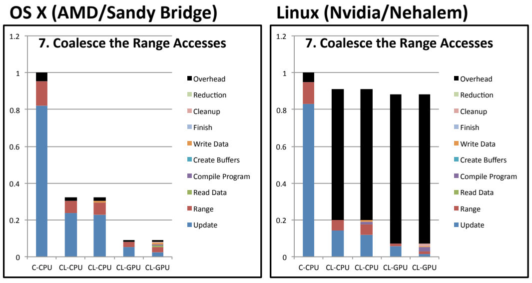
Comparing OS/Drivers

OS X(AMD/Sandy Bridge) **Linux (Nvidia/Nehalem)**



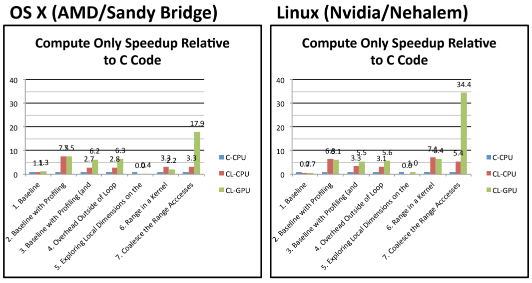
Comparing Final Results

OS X (AMD/Sandy Bridge) **Linux (Nvidia/Nehalem)**



Compute Only Performance

OS X (AMD/Sandy Bridge) **Linux (Nvidia/Nehalem)**



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?