Vivek Sivakumar
UID: 303652195

Lab 3 Report

Briefly explain the parallel strategies you applied in this lab. How is it parallelized? What is the expected communication overhead?

My first strategy was to get the sequential program working, this meant implementing all of the boiler plate that comes with OpenCL set-up: getting the platforms, getting the devices, creating the context, creating the command queues, creating the buffers, writing the buffers, creating the kernel, executing the kernel, and reading back the result. The steps were very close to the vecadd set-up except for buffers, where I had to create independently sized buffers for Cout, Cin, bias, and weight. I enqueued Cin, bias, and weight as write buffers, and I enqueued Cout as a read buffer for the result. My initial kernel was an exact copy of the sequential program with no optimizations for parallelism, this ran in ~90 secs.

My thought process for parallelizing the kernel was choosing the best approach for the CPU architecture. For the CPU the overhead of thread creation is quite high so I wanted to minimize that as much as possible. In the past matrix multiplication labs I parallelized the outermost loop because it meant the fewest number of threads and the largest amount of work per thread. The outermost for loops both run from 1 to 512 so it was quite simple to divide the work into 512/(# of work items) pieces, with each piece running on a different core. After doing this I saw immediate speedup at higher work items. I ultimately settled on 32 work items because it fully utilizes all cores and their hyper threading architecture.

The communication overhead comes from 2 main places: enqueuing write and read buffers, and dividing the outermost loop amongst work items. The communication for enqueueing write and read buffers is a one time write or read from the host program to the global memory of the device, the speed of which is completely determined by the bus from the host program's CPU to the device. Dividing up the work amongst work items has more possibilities, the most optimal in my opinion is a scatter approach where half the data is sent to processor 8 and processor 8 takes care of the second half of the nodes while processor 1 handles the first half (with the process repeating recursively until all processors have their data)

Evaluate your program in terms of the execution time. Please make a comparison with the serial version, and discuss the scalability of your parallel implementation using 1, 2, 4, 8, 16, 32 processors.

Sequential: 89.51 secs
1 work item: 81.137 secs
2 work items 46.966 secs
4 work items: 32.237 secs
8 work items: 19.536 secs
16 work items: 10.665 seconds

32 work items: 5.687 seconds

Looking at the data I saw a pretty close to linear speedup when doubling the the number of processors. I was surprised this program displayed such greater linear scalability than the matrix multiplication from the previous labs. I'm not sure exactly why this is the case but my best guess is that the parallel overhead for OpenCL is less than in MPI or OpenMP since most of the set-up for OpenCL occurs in the host program. The other possible explanation is that each processor just has more data to work on in this algorithm than in matrix multiplication, so the parallel computation dominates execution time resulting in a more linear scaling.

Challenges you encountered and how you solved them

The biggest challenge I faced in this lab was understanding the basics of OpenCL. Before I could begin writing any code I had to understand the flow of a program in the framework including all of the various steps that are required to set-up and run a program. I also didn't initially understand what a kernel is and why it was built within the host program itself so I had to figure that out as well. Finally I had to understand the memory model and the work sharing model of OpenCL, and how parallelization works within the framework.

However once I figured this out the coding and speedup was relatively straight-forward since this was on a CPU and I could use the same optimization principles from previous labs. I expect it to be more challenging from an optimization perspective as we move to the GPU and FPGA.

The final challenge I faced was actually calculating the execution time, using the given clock start/clock stop didn't work once I moved away from the sequential version because the execution was occurring on the device and not the host program. Instead I used the clEventProfiler in the OpenCL library to time the kernel execution, which is the most variable portion of the program.