Steven Smith

40057065

R. Jayakumar

Comp 426

Monday, November 30, 2020

# Simulation Project Design Documentation

## 1.0 Introduction

The project is another reiteration of the same problem we were tasked with solving in the three prior assignments. However, now we need to apply the proper multicore design patterns to appropriately separate the workload across the appropriate devices using the OpenCL C++ wrapper library. Throughout the document we will thoroughly discuss our development environment, task division, algorithmic changes from the previous assignments and memory optimizations as well as any troubles we may have encountered along the way.

### 1.1 Development Environment

The project was developed using a Linux workstation with the following specifications.

- OS: Ubuntu 20.04 LTS
- Linux Kernel: 5.6
- Processor: AMD Ryzen 7 2700 8 Cores 16 Threads @ 4.0 Ghz
- GPU: AMD Radeon RX 5700 8GB

- OpenCL version: 2.0
- POCL version: 1.20
- IDE: CLion
- Project Manager: Cmake 3.17
- Compiler: GCC 10.2
- Vcpkg: 2020.11 was used to acquire all packages used.

There were some minor hurdles in getting OpenCL to work and recognize all my devices. Namely, the use of an AMD processor made the setup process more complicated as AMD has discontinued support for its CPU OpenCL runtime, no longer recognizing a CPU as valid OpenCL target. Therefore, to be able to use my processor I installed the POCL platform runtime providing support for my CPU with a custom LLVM variant of OpenCL.

There are a few downsides from using POCL. Namely, it is slower by nearly a factor of two from the native implementation of OpenCL. Secondly, its version is not strictly a 1:1 representation of the OpenCL feature set. Version 1.20 of POCL has support for some features from the OpenCL 2.0 release. Thirdly, we cannot take for granted that both the POCL and GPU OpenCL platforms support the same extensions and features. This has caused a significant amount of headache and research to ensure that my game plan was still feasible. Thankfully, it was.

## 2.0 Task Division and Kernel Assignment

The program has four major tasks that it performs: Injection, Healthy Cell Mutation, Cancer Cell Mutation and Medical Cell Radial Expansion. A brief description of each task is outlined below:

1. Injection: This task randomly samples a point in the input space to use as the origin of a medical cell injection. Afterwards, the algorithm checks each of the valid injection point

and randomly decides if an injection will occur or not. This process is not data parallel but can be executed multiple times concurrently to have more than one injection per cycle.

2. Healthy Cell Mutation: The process of determining whether a healthy cell neighbours are more than majority cancer cells. If true, the healthy cell becomes a cancer cell. This is a data parallel task.
3. Cancer Cell Mutation: The process of determining whether a cancer cell neighbours are more than majority medical cells. If true, the cancer cell becomes healthy and the medical cells are consumed (returning to a healthy state). This is a data parallel task.
4. Medical Cell Radial Expansion: The radially outward movement of the unconsumed medical cells. This is a data parallel task.

Of the four task we mentioned, three of them are data parallel. This means that they work on a very large dataset. In our case the dataset is 1024*768 in size, nearly 800k. The same instruction is executed across those various data points. Consequently, Tasks two, three and four are appropriate to execute on a gpu with a large amount of Compute Units.

The Injection task on the other hand is not, it executes a single task on a single datapoint without a large amount of computation. This task can be executed in parallel and concurrently on different data points. However, the number of concurrent instances is limited due to practical simulation reasons. In our project, we use three concurrent instances of injection. Therefore, this task executes more quickly on a processor whose pipeline is more optimized for these kinds of operations and runs at a much higher clock speed.

To summarize, Task two, three and four will be executed on the GPU while task one will be executed on the processor. Furthermore, task two and three can be executed concurrently while task

four relies on task one, two and three having completed. Likewise, all gpu tasks must wait for all the injection processes to be completed.

Note: One task that is missing from the discussing is the graphics portion of the process. How is the vbo updated? At first, I was planning to use update it directly from the GPU but after facing some problems with the libraries required for my OpenGL implementation (Glew) causing compiler errors to occur within the cl2.hpp files needed for the OpenCL wrapper, the idea was abandoned. This results in a slight performance penalty as I update my vertex buffer object the same way as in the previous assignments. We are talking about a 2-3ms penalty hit.

## 3.0 Algorithmic Changes

In terms of the raw algorithm nothing changed. The algorithms are identical (logic-wise) as what we observed in Assignment 3. However, the conditional branches were needlessly long and confusing and had a lot of room to be shortened and cleaned up. We did just that. I also noticed that I was constantly rebuilding and determining the kernel parameters for the kernels within the OpenGL display loop. This is not necessary and harms performance of the application. In short, minor changes were done to code readability but almost nothing was altered when it comes to how the program is written. Task two and three were combined such that they run concurrently from the same kernel. The two kernels do the same comparison, the only difference is the outcome. Which outcome to select is purely based off the original color of the cell when the work item started execution. Therefore, this can be addressed using a simple conditional statement to make the Healthy and Cancer cell execute two different set of outcomes. This results in only ~700k passes over the data that accomplishes task two and three concurrently without having to traverse all these datapoints again and simply ignore the invalid cases.

## 4.0 Memory Layout Optimizations

It is without exaggeration that one of the costliest operations that can be performed is IO bound operations. In our case it would be the transfer to and from the host memory. As such one easy way to optimize our programs without making them device specific is to minimize our number of memory calls. This can be done by better handling what should be local or global memory. However, each mutation check requires access to a large subsample of the input space. One that would undoubtedly be too large for the local cache of the work groups. Therefore, moving content from global to local memory is not very feasible in our case.

What we can do, however, is limit the number of communications between the host and the target device. Fortunately, one of our target devices is the CPU which shares the same memory system as the host. Therefore, if we can determine how to share memory between the host and our gpu context than we effectively share memory between our two target devices. The solution to this is Shared Virtual Memory (SVM). This has been mentioned a few times already throughout the document and is enabled because my gpu has support for OpenCL 2.0, which from my understanding is a requirement for SVM support. Therefore, we allocated Shared Virtual Memory for our vertex buffer object source array, our array of direction index and our cancer and medical cell counters. This allows us to pass these pointers to our cpu and gpu and address them as if they were stored in our respective device's global memory. The reason for doing this is to eliminate the need to physically copy memory after the cpu and gpu queue executions. Doing this resulted in nearly a 3X speedup in the performance of the program as well as help alleviate the weird anomaly that we reported during assignment 3.

Unlike Assignment 3, instead of using TBB to execute the Injection the OpenCL EnqueueNativeKernel function is used to execute our native C++ code onto our processor and accessing our SVM data structures directly.

## 5.0 Conclusion

In conclusion our we executed task two and three concurrently one the gpu while task four was executed sequentially since it needs to wait on the results from task two and three to be completed to maintain the integrity of the data. All three tasks execute the same instruction on over 700k datapoints. As such it is most suited to be executed on high CU devices.

Task one, Injection, is an example of an instruction executed on a single dataset that is randomly sampled from the input space. This task can be running multiple times concurrently, but never in a large number as it is impractical for the simulation. Three concurrent instances of Injection are used to inject medical cells into the simulation on every cycle. Because this task is not reliant on traversing many data points or computations it is not easily split to run in parallel, it only checks for conditions. As such a processor with its higher core clock and optimized pipeline for complex instructions is more suited to perform these tasks.

Shared Virtual Memory is a neat feature introduced in OpenCL 2.0 that is supported by my GPU and used in this instance to drastically improve performance by nearly 300% by reducing the number of complete memory transfers to/from the host and devices.

Lastly, due to some issue with the glue library and incompatibilities with cl2.hpp we were not able to setup the use of opengl and OpenCL Interoperability feature that would have aided us to optimize the program even further and would have helped shave off around 3ms. Albeit, given the performance which given the average 8ms execution time is not an insignificant margin. Unfortunately,

as mentioned I was not capable of getting it to work without my compiler screaming about errors within the OpenCL c++ wrapper.