**Chapter 5: Conclusion**

CHRISTEN FORD, Baldwin Wallace University

1. **Conclusion**

This thesis showcased three-dimensional RRT and a GPU accelerated variant of RRT. We sought to determine whether a GPU accelerated version of RRT could outperform its unaccelerated counterpart. To realize this, we utilized the Project Malmo framework by Microsoft Research as a basis to work from. We then built a world and collision detection system for the agents to work with. This was necessary as the level of information the agents needed to function was not easily retrievable from the framework. After this, we created four maps for our agents to path across. Using A\* as a benchmarking algorithm, we then proceeded to gather data on RRT and RRT-GPU regarding run time, path length, number of heading changes, and total degrees in the path. This was repeated 1,000 times to generate an appropriate population to for statistical testing. The results of our testing show that, in our specific scenarios, accelerating RRT does not yield significant improvements upon run time. This was antithetical to what we sought to achieve. Nonetheless it provided valuable insight into the nature of RRT as a viable path planning algorithm.

In a practical sense, RRT must know of its immediate surroundings. In this thesis, our maps were small enough to allow us to load all geometry data for a given map into the agent. Realistically, RRT would also need a scanning function to allow it to dynamically load and unload immediately surrounding world geometry data. RRT utilizes a global node list and as such was not amicable to parallelization. Much of the overhead incurred with RRT-GPU stems from the translation between the data containers used by Numpy and the data containers used by PyCuda. In addition, the size of our data set, and the parameters we chose to use with RRT-GPU along with proper Cuda practices meant that the algorithm was not guaranteed to even run on the GPU in every instance. These factors translated into a slight increase in run time of RRT-GPU compared to RRT.

Given what RRT and RRT-GPU set out to achieve, we can conclude that in two-dimensions both algorithms perform well enough to act as suitable path planning algorithms. Their randomized nature means that unless post-smoothing is applied after the path is generated, the path will not be minimally optimal. Likewise, paths generated by both algorithms are not realistic. Both minimal path length and path realism are exceedingly important in applications involving computer graphics or motion planning.

For applications that involve real-time motion planning RRT and RRT-GPU are significantly faster than A\* for numerous factors. First, A\* must evaluate all adjacent neighbors (that have not already been seen or evaluated) before choosing the best cell for evaluation. RRT does not need to do this, instead it samples randomly form a predefined sampling space and branches to a nearby node. RRT has no evaluation requirement of neighbors. Second, unlike A\* which must move from neighbor to neighbor, RRT can jump around the search space. Paths generated by RRT tend to contain significantly less nodes as a result. Likewise, we can bias RRT with a slight probability to sample the goal immediately. This means that at each iteration RRT has a small chance to instantly find the goal, skipping possibly many, intermediary steps.

RRT does have drawbacks to its use. First and foremost, it requires full knowledge of the search space. A\* can naturally discover the search space as it proceeds through its algorithm. For RRT to exhibit the same behavior, it must be modified with a localized scanning algorithm to force it to sample only from discovered space. This is in fact what RT-RRT\* does with its ellipsis sampling algorithm.

1. **Future Work**
2. **What I Learned from This Experience**

This thesis revolves around computational geometry and high-performance computing. I utilized scientific computing libraries such as NumPy, PyCuda, and Pandas. All the software packages utilized in this thesis carry real world relevance. NumPy and Pandas are used daily in professional environments to analyze vast quantities of data for varied purposes. As a wrapper for Nvidias’ CUDA runtime environment, PyCuda allows users to complete simultaneous complex transformations on data utilizing a discrete graphics card in a fraction of the time it would take a central processing unit. This has proven a valuable experience in gaining insight into the uses of some of the most commonly used libraries in the Python programming environment.

More importantly, this thesis helped enriched my knowledge on parallel programming. This type of programming differs from sequential programming so much that it took me roughly four months just to understand the basics around it. Thankfully, PyCuda saved me from many of the messy underpinnings of working within this paradigm. My first attempts at parallelization were crude manipulations of Nvidia sample programs. As I continued my foray into writing CUDA code by hand, I realized that there had to be a better way. I eventually stumbled upon a set of functions within the PyCuda documentation that encapsulate almost all the setup for accessing the data and allowed me to simply pass the graphics card the data I wanted to work with and a function to apply to it.

The issue then shifted to representing that data in a meaningful way so that I could allow the agent to utilize it. I ended up utilizing PyCudas built in vector objects to pass multiple data points in a single object. This worked well enough, but as mentioned earlier in the thesis, this technique was a contributor to the increase in run time of RRT-GPU over RRT. The other contributing factor was the translation between sending data to the graphics card and then receiving it and translating it back.