Gaming Laboratories International

Colorado School of Mines Field Session 2018

Victoria Girkins, Evan Bergo, Savannah Paul & Paul Reimann

19 June 2018

Table of Contents

Introduction

Gaming Laboratories, International runs simulations of slot machines to verify expected behavior and returns. These simulations must go through 10 billion iterations for the game to be legally verified correct in all states. Running this many simulations can take anywhere from a few hours to two weeks depending on the complexity of the game, which leads to slower progress and less efficient work. This warrants the need for a faster method of running these simulations. This project's goal is to provide a proof of concept that running slot simulations on a graphics processing unit (GPU), rather than solely on a CPU, would dramatically increase the execution time.

Our product vision was to create three versions of a slot machine simulation to compare the speeds when using linear techniques on the CPU written in C++, using multithreaded techniques on a CPU, and using multithreading techniques on an nvidia GPU written in CUDA. All versions of the simulation must adhere to the predetermined return to player (RTP) ratio set by GLI, which represents the percentage of wagered money that is returned to the player by the slot machine, over the course of 10 billion games. Note that obtaining a correct RTP is a way of verifying the correctness of our simulations, but is not the end goal.

Requirements

Functional Requirements

- Create a linear, brute force, CPU-based simulation
- Create and optimize a multithreaded CPU-based simulation
- Create and optimize a parallel GPU-based simulation
- Gather data on speeds of each version to run 10 billion simulations
    - Use multiple machines with differing processors to monitor this effect
- Present data in a way that makes it easy to compare and interpret results

Non-Functional Requirements

- Write in C++ for CPU code and CUDA for GPU code
- Deploy on GLI's sole graphics card (Nvidia GeForce GTX 1080 GPU)
- Accurately represent the given theoretical slot machine by ensuring the resulting data follows the RTP set by GLI
- Accurately generate random numbers, adhering to GLI standards
- Make the code base scalable so that extra game features may be easily implemented
- Optimize GPU code by utilizing all available cores and managing memory efficiently
- Parallelize different aspects of code and compare methods
- Create code that is well-documented so that GLI may continue the project

System Architecture

The main difference between the CPU versions (versions 1 & 2) and the GPU version (version 3) is where the data is being passed and processed in the hardware and in memory. Versions one and two are run entirely on the CPU; as seen in Figure 1, data is passed between the CPU's processor and memory. In version three, the simulations are run on both the CPU and GPU as seen in Figure 2. The data starts on the CPU until it is copied over to the GPU to execute the simulation and manipulate the data using multiple processors, once the simulations are done the data is copied back to the CPU to be read by the user.

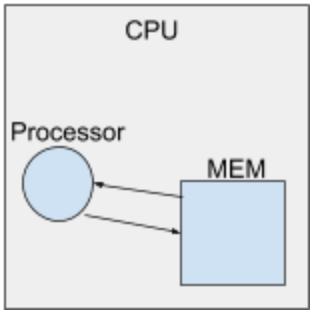

Figure 1: Data Flow Versions 1 & 2

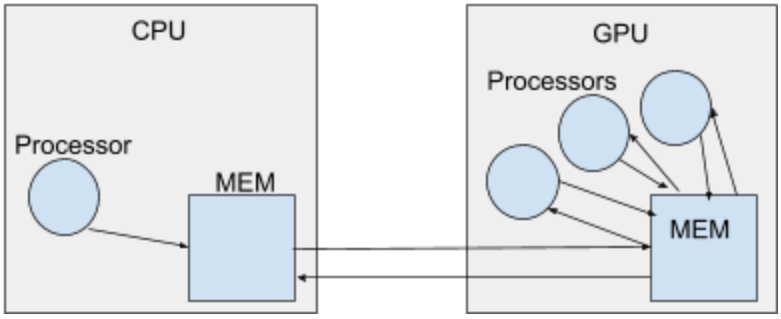

Figure 2: Data Flow GPU Version

Each slot machine game, no matter how complex, requires the same basic steps in order to play. The user inserts money to place bets on lines, the user pulls the "lever" to spin each reel thereby creating a new board state, the current board state is evaluated by the machine and money and granted to the player based on symbol distributions. Figure 3 shows how we represented this general game flow in our code using a series of functions that are all to be called by an overarching driver function.

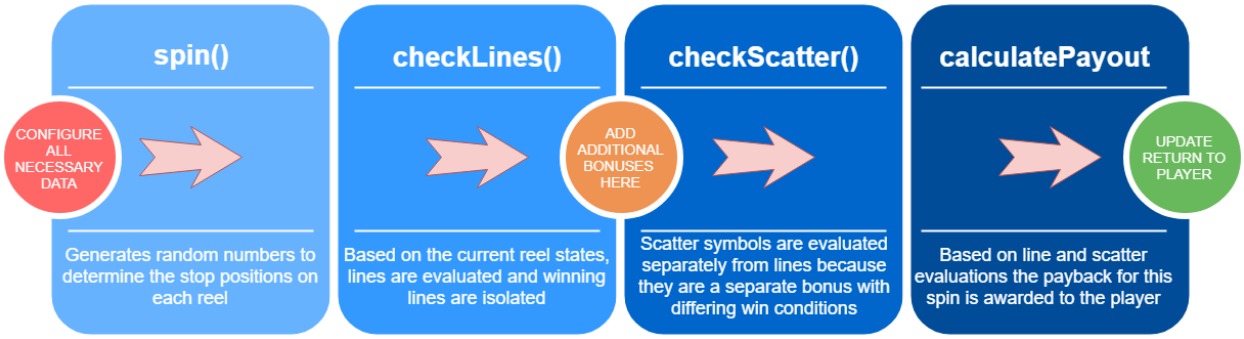

Figure 3: Core Slot Game Functionality

By manipulating these core functions we were able to design different methods of simulating the same slot machine game. Figure 4 below shows the version history relationship tree of all the versions and methods of our slot machine game.

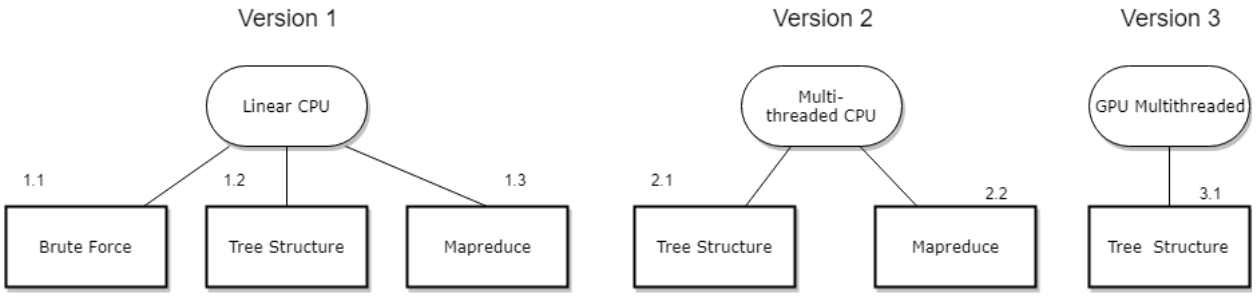

Figure 4: Version Tree

Each method version is a variation of the previous only with the more advanced simulation techniques applied and integrated into the code while each method within each version is a different approach to evaluate the paylines in our game. As we continued throughout our project we narrowed our focus to determine the optimal method for GLI to run the simulations on.

Version 1 - Linear CPU

We designed three different methods to model the linear slot machine game: the brute force, tree method, and mapreduce. On each iteration, the brute force method will generate a stop position for each reel, project the current reel positions onto each payline, check the current reel states for scatters, and then evaluate each payline to determine if there is a payout. This is considered a "brute force" approach because each payline is being manually projected and evaluated at each new board state, which is a completely unoptimized approach.

The tree method generates a tree structure to represent all of the possible paylines. This tree combines similar features in paylines so that it is possible to check multiple lines at the same time. For example, if paylines 1 and 2 both include the first and second top row symbols, then they can be eliminated at the same time if the top two symbols on the current board state are different. On every iteration, the board state will be evaluated against this tree structure to determine which lines grant a payout. This saves time because we are able to evaluate all the paylines at the same time, in the most efficient way.

The map reduce method generates all the possible outcomes and reel combinations in advance. On every iteration the randomly generated spin, instead of mapping to a reel stop location, it maps to one of the predetermined outcomes with a predetermined payout. This saves time because outcomes are already determined to be a win or a not so there is no longer a need to actually evaluate each line. For version 1 the fastest method we were able to implement and

optimize was the mapreduce method, which follows the function structure as seen below in Figure 5.

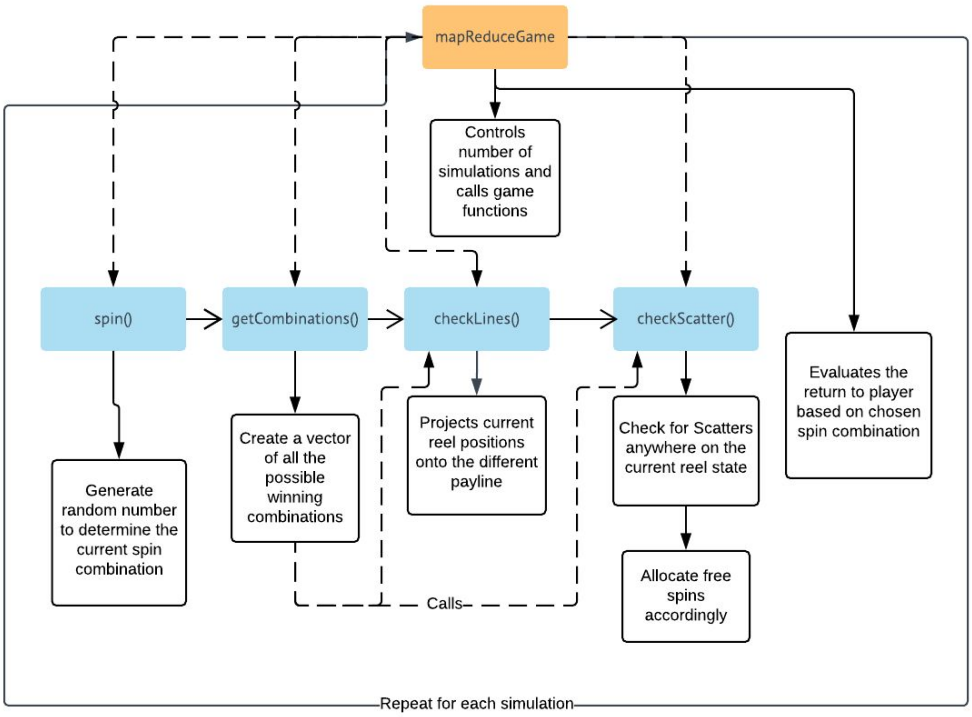

Figure 5: Mapreduce Method Design

Version 2 - Multi-Threaded CPU

In this stage we continued with both the mapreduce method and tree structure and experimented with parallelizing these in different ways in order to find the maximum speedup.

With the mapreduce method our most optimal version multithreads the preprocessing step that generates all the possible combinations and that multithreads by each simulation.
The tree structure method executes all of the preprocessing steps then multithreads the total number of simulations.

During this stage we experimented with the optimal number of threads on which to run our simulation; creating too many threads can be inefficient and slow down the whole process, while using too few threads may fail to utilize the computer's full potential. Through experimentation, we found that 16 threads is the ideal number on which to run the simulations from the processors we tested. However, other processors on the market capable of computing more than 16 concurrent threads would benefit from a higher program thread count. The slowdown resulting from a large number of threads (ex: 1024) is minimal compared to having too few threads for the processor.

The fastest implementation at this step was the mapreduce method; however, GLI is looking for a solution that will ideally be implemented by their company to aid with their simulation testing and the mapreduce method is not ideal for this type of testing.

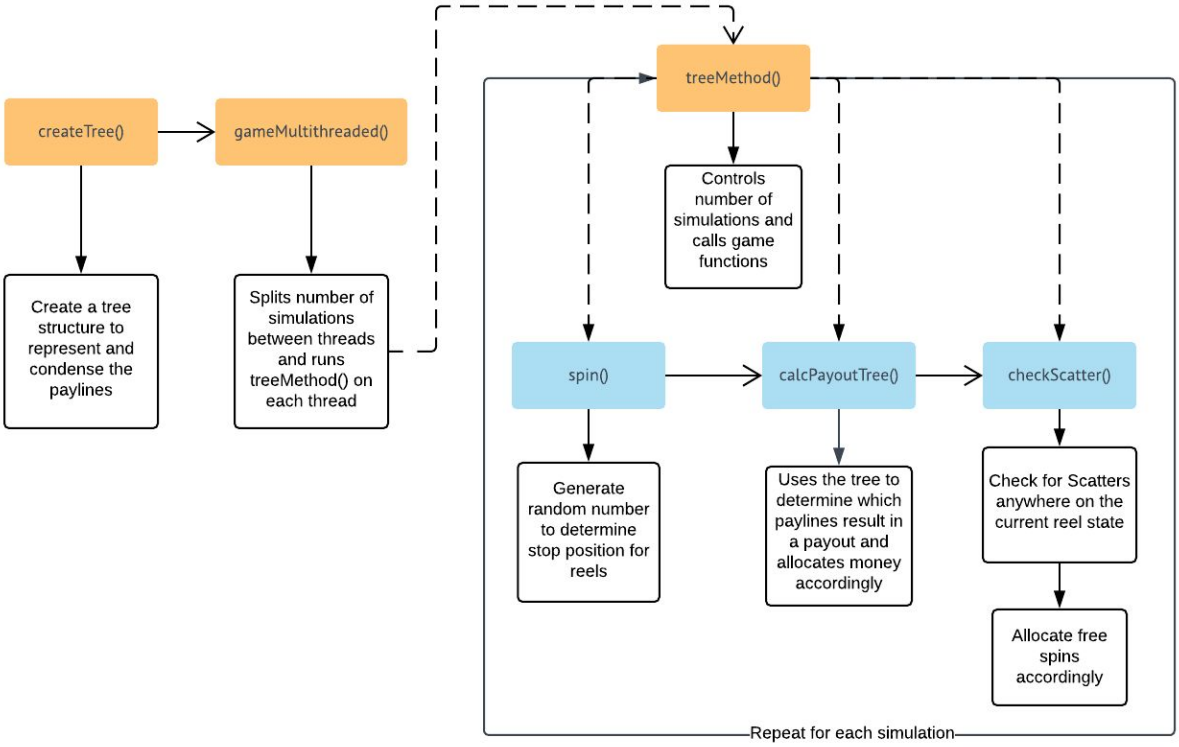

Figure 6: Tree Structure Method Design

With this in mind, we had to consider that most, if not all, of the games GLI deals with have far more reels and reel stops than the simplified version we coded. The mapreduce method's efficiency does not scale with larger games since increasing the reel size from our practice game would dramatically increase the execution time. On the other hand, the efficiency of the tree structure method is not affected as drastically by an increase in game size. This is why we chose to continue with only the tree method in stage three.

Technical Design

During stage three, we translated our CPU multi-threaded tree structure code into CUDA code that can be run on a GPU. The main difference between this step and the last was that we needed to manually allocate the memory needed on the GPU before we called the function. In order to utilize the GPU fully, we included code that would determine the optimum number of blocks for that machine. We found the maximum block size to be 256 threads. From here, we simply decided which game functions to run on the GPU instead of the CPU and modify them to CUDA standards.

A GPU has many different types of memory with different purposes. The global memory is available to all threads on the GPU and can be accessed concurrently. While this contains the greatest amount of memory (usually several GB), it is also the slowest to access. Another useful type of memory is called shared memory. This is memory that all threads in a block can access.

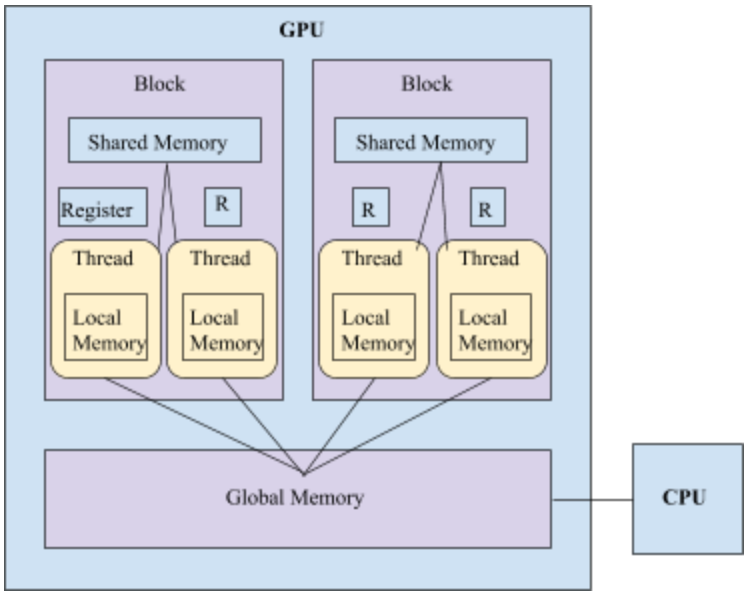

Figure 7: Memory Hierarchy Within a GPU

This is many times faster than global memory, but is limited to 40 KB and is only accessible to threads in a single block. Other memory types are automatically optimized by the cuda compiler. Optimizing the code involves moving as many variables as possible to faster memory locations. When optimizing our GPU code, we found that storing some variables on shared memory as opposed to global memory leads to substantial speedup. Putting frequently-used, smaller-sized variables into shared memory decreases the access time for these values and causes additional speedup.

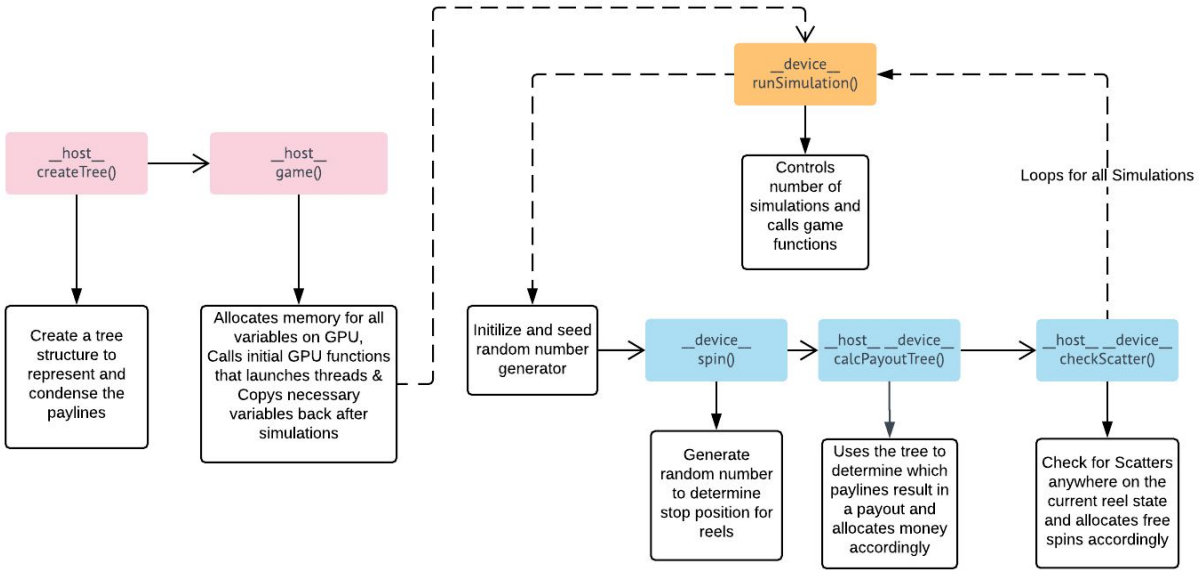

Figure 8: GPU Tree Structure Design

An additional optimization resulted from using a stack to traverse the tree structure rather than a queue. There are several reasons for the increased speedup. First, using a stack requires less total memory. Because of the way the tree is structured, a depth-first search algorithm using a stack results in less explored nodes at any given time, reducing memory usage. Additionally, a stack is easier for the internal caches to work with since the most frequently accessed addresses are around the starting location of the stack whereas the beginning of a queue with constantly be changing as it pushes and pops data. This makes caching difficult since there is no constant memory location used every simulation and the memory is less likely to be stored or used in the cache.

Another optimization resulted from the random number generator (RNG) used. A good RNG must be relatively fast and have a long period, which is the amount of numbers that can be generated before the sequence of random numbers repeats. The standard RNG algorithm used by GLI is called the Mersenne Twister, which has a period of $2^{19937}-1$. The Cuda random libraries include this method, but only allows 256 threads in each block to access the library concurrently and up to 200 separate sequences of the RNG. These limitations suggested a faster, better RNG was needed with faster generation. We therefore implemented the XORWOW random library which is faster and has a period of 2^{190}, which is sufficiently long considering we are implementing only 10 billion ($\sim 2^{34}$) simulations that call the RNG one to two times per simulation.

Decisions

GLI made our programming-language choice for us by their requirement that the GPU version be able to run on their nvidia GeForce GTX 1080 GPU. This specific hardware can only be programmed in CUDA; therefore, we chose C++ for versions one and two since CUDA follows C syntax. This made it easier to reuse code and to relate the versions.

Scalability was another important aspect of our code. To be scalable, our code base needed to be easily understandable and extendable, and adjustable to different slot-machine types. To help achieve these goals, our code draws all information about the game from configuration files. While slot machine games may differ in size and bonuses, they all share the same common features: a set of reels with differing symbols and distributions, a set amount of symbols that can be used throughout the game, a set of paylines that represent lines on which the user can bet, and the payback amounts for each type of win. It was due to these cross-games similarities that we chose to populate these variables from configuration files; in this way, different games can be simulated without changing any of the actual code. To account for future games that may have special features and bonus rounds/reels, we partitioned our game into separate functions that divided the core functionality. This helps accomplish the goal of extendability. The main functions of our game, for each version, are `spin()`, `getCombinations()`, `checkLines()`, and `checkScatter()`. Any version of the game will have to use these functions in some way, since they represent core properties of a slot machine.

The need to run 10 billion simulations is fundamental to our project given our original problem statement. The magnitude of the problem size inevitably leads to large variables and data structures. Because of this, we represented most integers with either `uint32_t` or `uint64_t` data types, which store much larger values than the `int`.

When designing the multithreaded version on the CPU, we used the `atomic` library in order to reduce the competition of resources between threads. When the threads are launched, they will all attempt to access and update the same common variables that track the game statistics. This can cause problems with loss of data when two threads try to access the same variable at the same time. To fix this, we used the `atomic` library which allowed us to create `atomic` variables which are variables that force threads to “take turns” when accessing them. However, having to constantly pause all the threads whenever there’s contention caused our program to slow down considerably. Ultimately we tried to use `atomic` variables as little as possible and alternatively use local variables that we can combine at the end.

GPU programming on CUDA only supports primitive data types, meaning that high-level data structures like vectors and maps cannot be passed to the GPU’s memory. This why, on our GPU

version, we changed all variables into arrays or pointers (low-level data types which can be passed between host and device). This allowed function calls to be made to and from the GPU. CUDA also does not support the same random number generator as C++. This required us to use an external library, CURAND. There is no "best" RNG; different random functions in the CURAND library behave differently and are ideal for different applications. The mersenne twister was the first option, but thread synchronization would be necessary to ensure too many threads did not access the randomly generated numbers at one time. Instead, we used the XORWOW function of this library, which allows any number of threads to concurrently access and generate pseudorandom numbers with a high period.

Results

We implemented all the requirements asked by our employer, and we designed a linear CPU-based game, a multithreaded CPU version, and a multithreaded GPU-based game. Our goal was to show the speedup that can be gained when simulations are run in parallel on a GPU.

In order to ensure our game versions were mathematically equivalent to the sample game we were given, we tracked how much money was inputted to the game and how much was returned. By tracking the monetary return generated by each type of win (line, bonus and scatter), we can determine the RTP of our game and verify that it matches the data from the sample game. To demonstrate and prove that there was definitive speedup between versions, we not only ran and timed the minimum 10 billion simulations on the provided machine at GLI but also obtained benchmarks from a variety of computers to obtain data for an assortment of processors and graphics cards.

Our testing showed that on the computer provided by GLI, the linear CPU game using the tree method took 78 minutes and the CPU parallel version took 11 minutes, about a 6.8X speedup. The GPU game took just 24 seconds, a 27.6X speedup from the multithreaded CPU game. This is an overall speedup of 187.7X from the linear version to the GPU version. We proved that running the simulations on a GPU utilizing all the possible threads was the fastest method of executing 10 billion simulations as illustrated by Figure 7.

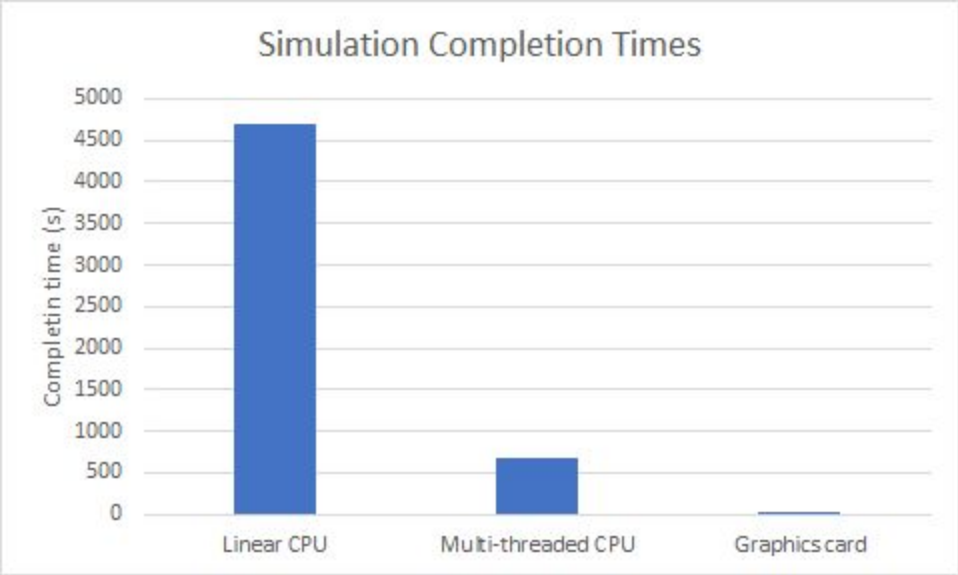

Figure 9: GPU Tree Method Results

There are many different types of slot machines that include more complex functionality. Examples include bonuses that trigger entirely different reels with different symbol combinations and weights, and games with many more reels, which may be extremely long. To extend the code to encapsulate this extra functionality, independent bonus functions can be made and added to the `SlotGame` class to be called by the game driver function when appropriate. We did not have time to implement any versions of the game other than the traditional version that was initially given to us. However, in order to make the code base more useful to GLI and ready for future editing, we included a series of extra features.

We added the ability for the user to enter in the number of simulations he or she would like to run. We also included a manual "debugger mode" which allows the user to run the game one simulation at a time, displaying the current board state and payout amounts after each spin. The debugger mode also includes a feature that allows the user to pick the stop location for each reel, essentially eliminating the random aspect of the game so that the user may control the outcome of each spin. This will help GLI to inspect the internal workings of the game and to add more complex features with ease.

Lessons Learned:

- Only primitive data types can be used on a GPU, so only arrays and similarly simple containers can be accessed. We also found that using only arrays in the CPU versions reduced memory access times and further sped up the code.
- There are different types of memory within the GPU. Global memory can be seen and accessed by all the blocks; shared memory can be seen by all the threads in a given block but cannot be seen from outside the block. We were initially storing everything in global

memory until we found that putting relatively small, frequently-used variables into shared memory dramatically reduced our GPU runtime.

- When multithreading on the CPU, there are many different libraries that can be used. The `atomic` library allows variables to be accessed across multiple threads without resource contention. We initially were using this library to summarize thread results about game statistics into aggregate results, but we found that the `atomic` library significantly increases the time needed to access and write to memory. Since we write to multiple variables on each thread for every simulation, this resultant slowdown was significant and we found that the `atomic` library was not optimal for our program.
- Consistent meetings with our client company proved extremely helpful. Every time we met, they helped clarify an issue or gave us more detail on some aspect of slot machines. Had we failed to meet often, basing our project only off the initial requirements and detail GLI gave us, we would have ended up with a lower-quality product. It was through these meetings that we learned how to perform a map reduction, how slot-machine games might vary and therefore how to make the code flexible for future use, and what sort of tolerance was reasonable for evaluating our RTP for correctness, to name a few. This pattern of open communication paved the way for a robust implementation of the product. If we had doubts about the usefulness of constant interaction with clients, we don't anymore.

Addendum

1. Figures 1, 2, and 7 were formatted incorrectly because there was a problem with downloading Google Docs drawings to PDFs that made them distorted. So we just found another way to download our report that wouldn't affect our images
2. Explanation of the atomic library was addressed by including in the decisions section more on the atomic library and what use we get from it (making atomic variables).
3. "Experiment simulations" was just misleading wording on our part, they are the same type of simulations as all the others were running. So we cleared up that wording to make that point more obvious.
4. We took out a paragraph about memory semantics in order to focus only on the types of memory that are relevant to our program.
5. We added more detail to our lessons learned to clarify some aspect that were lacking and added a point about company communication.
6. Other formatting issues were addressed to make our report more aesthetically pleasing as well as grammar and phrasing corrections.
7. We didn't apply the suggestion to add more about GPU scalability because this was not the main focus of our project, we were only given one GPU to run our program on and it is the GPU that GLI will be using in the future to run our work.