Parallelizing Genetic Algorithms Project Report

Raymond Chee

Summary:

This project explores the design space for parallelizing generic genetic algorithms. I implemented both a sequential and a parallel version of a genetic algorithm optimizing a simple fitness function and compared the performance between the two implementations. The parallel version targets the GPUs on the GHC machines.

Background:

Genetic algorithms are a class of optimization functions inspired by genetic evolution from biology. Each group of weights to optimize is called an individual of the population. The weights themselves are referred to as genes. A genetic algorithm proceeds as follows: First, the input population's fitness is evaluated, where the fitness function takes a population and returns a numeric score for each individual that characterizes how "fit" the individual is. This is the value we are trying to optimize, so the exact details of the fitness function is application-specific. Then, we evolve the population over a fixed number of generations. Each generation produces a new population from the older parent population and evaluates the fitness of the new population. If the fitness of the population converges or otherwise meets some other stopping criteria, we can terminate early.

Creating a new generation involves randomly choosing “parent” individuals to combine and form new children individuals. Parents with higher fitness scores are more likely to be chosen, which should drive the population to an optimal value. Combining parent individuals produces a new set of genes using a genetic operator applied to the old parents. A common example of a genetic operator is the crossover function, which chooses a position in the gene vector and copies each gene to the left of that position from the first parent into the child, and copies the genes to the right of that position from the second parent into the child. Finally, the child’s genes could randomly “mutate”, which involves flipping a bit in the gene so that the children are not simply a combination of the parent’s genes. The random initialization and mutation feature in genetic algorithms allow it to avoid converging onto local optimas, which many other optimization algorithms tend to suffer from.

The key data structures in this algorithm are the arrays of individuals and their gene arrays. In my implementation, I defined individuals as an array of bits, where each gene is a single bit in the array. I used a fitness function that simply adds the values of the individual’s genes together, but the algorithm is agnostic to the particular fitness function used since it doesn’t try to parallelize its evaluation. I chose the crossover function as described in the previous paragraph as my genetic operator, and the method I used to randomly choose parents is to weight each individual by its fitness and randomly sample from this weighted distribution.

With these parameters, the workload is the following:

- For each generation:
    - Create a prefix sum array of the fitnesses of the population
    - Create the children array:
        - Choose 2 random value between 0 and the total fitness of the population and find the indices of the largest values in the array that's less than the randomly chosen values. These are the parents of the children
        - Perform the crossover operation on the parents and for each gene in the child, randomly flip the bit with some probability.
    - For each child, evaluate their fitness (sum all of their genes together).

Creating the children array and evaluating fitnesses is trivially parallelizable since the children are not related to each other. However, creating a prefix sum array is a sequential task by nature. Additionally, the nested format of the data structures makes it difficult to directly port over to the GPU.

Approach:

Data structure-wise, I ended up flattening the nested gene array into a single, large array, and made the individuals store the starting index in this large array instead of a pointer to its own array. The first part of my approach involves using the exclusive scan algorithm to produce the prefix sum array. To create the children array, I divided the children array into chunks, and had each thread create their subset of the children.

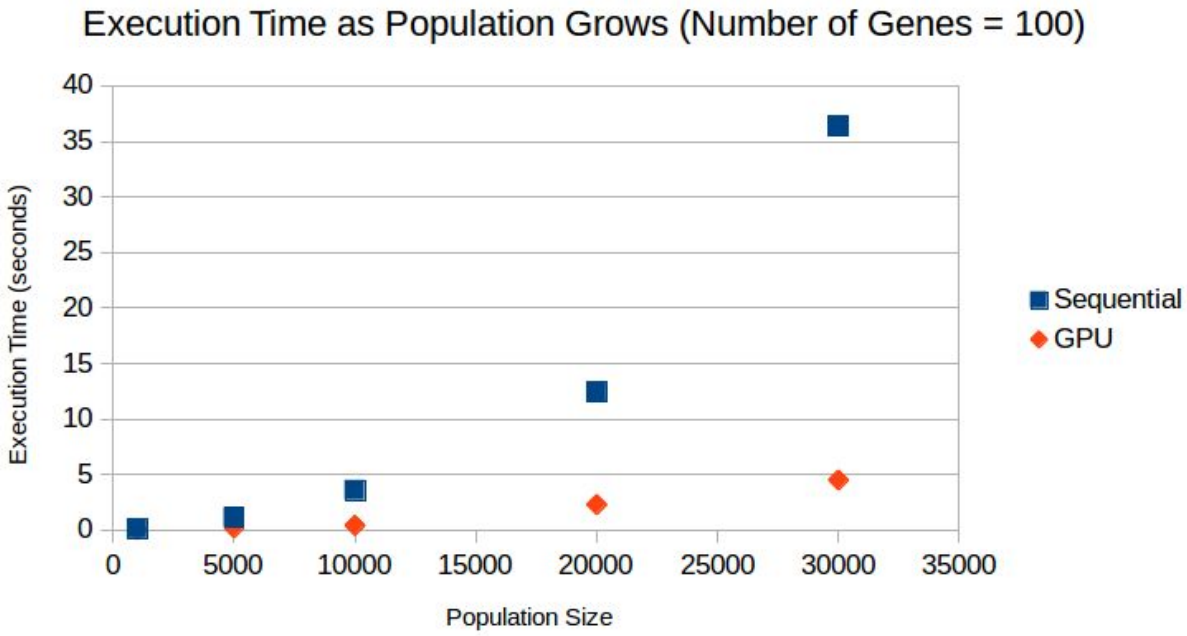
Similarly, for the fitness evaluation step, I divided the set of children between threads and had each thread run the fitness evaluation function for each child. Because each step depends on the previous step, I synchronized the threads after each step completed. Additionally, since synchronization between blocks was required for correctness, and the version of CUDA installed on the GHC machine does not support grid synchronization, this limited my parallelism to having a single block and 512 threads in that block.

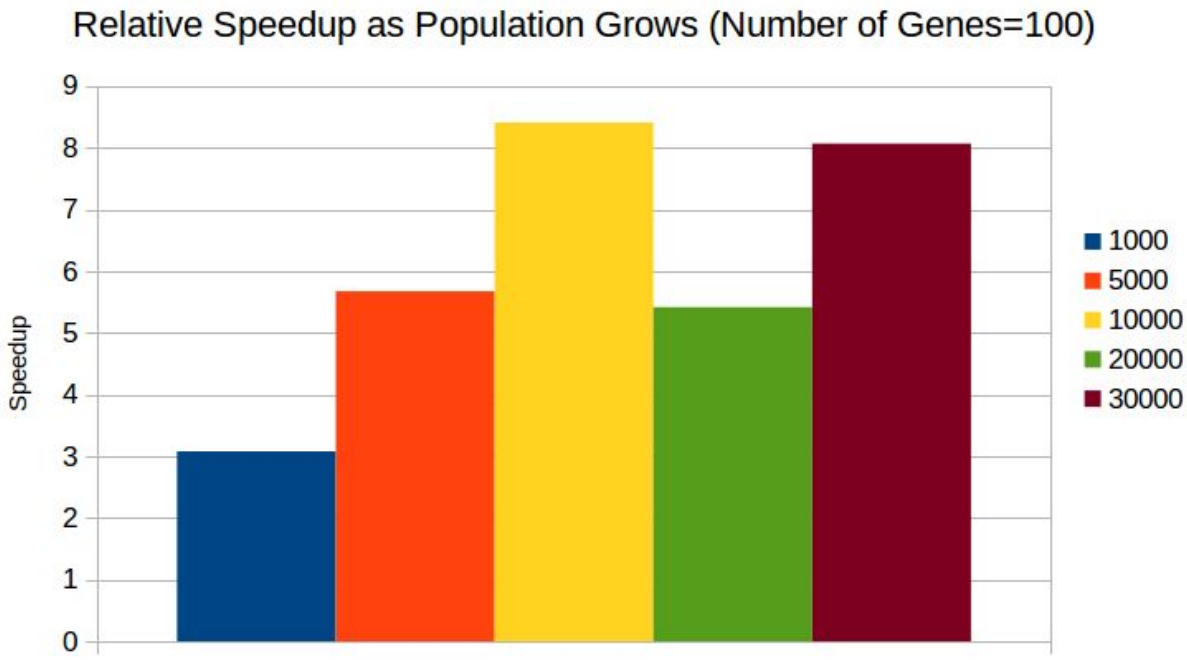
A previous iteration of this code parallelized everything except for the prefix sum. In this version, the time it took to execute was comparable to the sequential version, achieving only a 1.11x speedup.

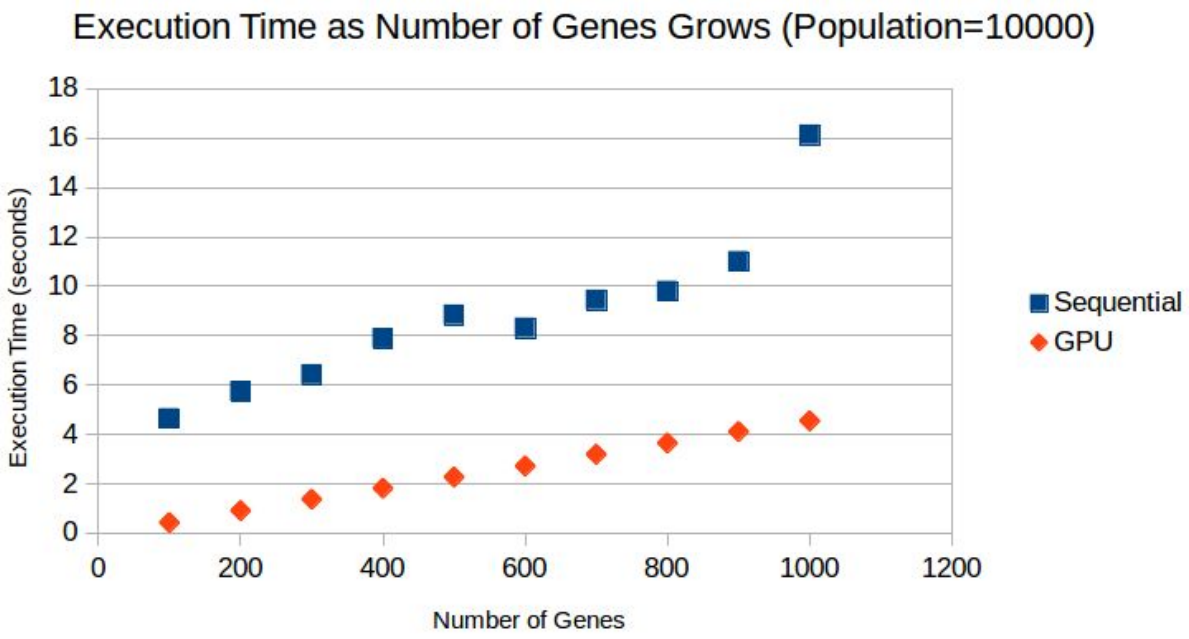
My current implementation implements this in a single kernel launch. Previous iterations involved multiple kernel launches for each step of the algorithm, but the overhead from these kernel launches caused my code to run too slowly to terminate in a reasonable amount of time. However, this was before I implemented the exclusive scan, so if the runtime characteristics of this approach with exclusive scan would improve.

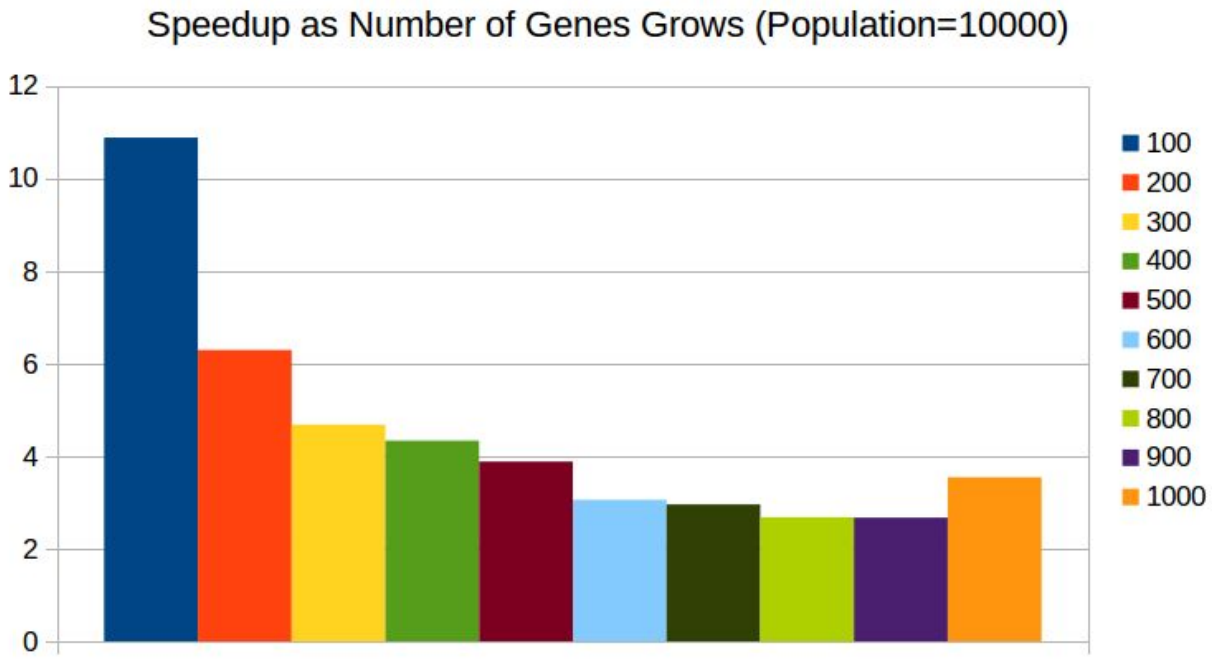
Results:

I measured performance by measuring the time it took for the main algorithm to run in both the sequential and GPU versions of the code. This excludes setup time. Then, I computed the speedups that the GPU version achieved relative to the sequential CPU version. To produce these results, I ran the code on ghc28.ghc.andrew.cmu.edu.


Execution Time as Population Grows (Number of Genes = 100)
Execution Time (seconds)
40
35
30
25
20
15
10
5
0
0
5000
10000
15000
20000
25000
30000
35000
Population Size
Sequential
GPU


Relative Speedup as Population Grows (Number of Genes=100)
Speedup
9
8
7
6
5
4
3
2
1
0
1000
5000
10000
20000
30000


Execution Time as Number of Genes Grows (Population=10000)
Execution Time (seconds)
18
16
14
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
Number of Genes
Sequential
GPU


Speedup as Number of Genes Grows (Population=10000)
12
10
8
6
4
2
0
100
200
300
400
500
600
700
800
900
1000

With 512 threads, I did not get a 512x speedup. The differences were likely due to the memory bandwidth to copy results back and forth, as well as the synchronization required between threads.