CS365 Assignment 05: Genetic Mimicry of Letters

Marie Kiley
Andrew Georgiou
York College of Pennsylvania
May 2019

Github repository: https://github.com/AlaskaJay/YCP_CS365_Assign05

For this project, we used a genetic algorithm to train a program to find the best parameters for creating letters. In order to compare the speed of the algorithm, we developed both a sequential and a parallel version of the algorithm. The parallelization of the program was then done using the CUDA API. At its heart, the problem is fairly simple; For each generation, develop a fitness function which determines how close the current generation grid is to a comparable grid which contains the values of the letter that the program is trying to replicate. The function will then either add or subtract some arbitrary number as either a 'reward' for being right or a 'punishment' for being wrong. Once a fitness value is determined for each generation, the program will then sort the values from least to greatest and then mutate children based on the most successful of each generation. This should allow for the program to compute the best possible fitness ratio and thus breed the same letter as that that is shared on the comparable grid. In order to parallelize this, we simply parallelized the fitness function as it was the most computationally intensive part of the entire algorithm.

To tackle the problem of creating a genetic algorithm, we had to break the process into separate steps; the two major steps being the fitness function and the mutation function which measure a generation and create the next generation respectively. In both versions, the program first initializes the fixed boolean array which represents the letter, then randomly assigns values to all of the generators. It should be noted that both implementations use a 2d array of fitness values (floats) and a 2d array of seed values (floats) to represent fitness for each generator and the likelihood of a space being black or white on each pixel for a generator respectively instead of a generator struct. This is to provide ease for parallelization with CUDA, similar to

Assignment 4’s structure moving away from the use of a particle struct. Most generators at the beginning are wildly incorrect and the program has no ability to create anything meaningful. The program must then use the fitness function to evaluate and sort the generators, so that only the best generators are used for the next generation. This is the point where the two implementations differ. The sequential version uses a loop to iterate through all generators. The half of the generators with the best fitness value are then used to create the next generation. The mutate function works by taking a singular index that represents a generator and then creates two children generators that are close, but not identical to the parent or each other. This new set of generators made by the mutation function are then used as the next generation. It should be noted that quicksort is used to sort the generators.

After implementing this algorithm as both sequential and parallel, we have found that only a sufficiently arduous fitness algorithm is befitting of parallelization. Additionally, due to the way the mutation function is implemented and the quirks of quicksort, the first generation tends to be faster than the rest, with the second generation being second fastest. With our initial array size of 8 by 8, the fitness function was relatively simple. This lead to increasing the array size tremendously. The first increase in size we tried was from the original to a 64 by 64 array, randomly generated instead of taken from an array. Ass the proof that the algorithm can replicate an array of arbitrary size, it was unnecessary for us to spend additional effort creating hard coded arrays to test against of larger sizes. This was continued for multiple powers of two, with each tick duration being measured. A 512 by 512 pixel array was found to be on the edge of being too arduous for the sequential version to do so in any reasonable time with 1000 generators. These timed to measure the tick duration are stored in separate files in a folder called results in the project folder, organized by array size, where it is clearly demonstrated that the array size creates large differences in fitness time.

Throughout this project, we have learned that genetic algorithms present unique implementation problems that must be solved creatively. We have found out that implementing them in parallel is very difficult however is very rewarding after those difficulties are overcome. As shown above, the fitness function is the most computationally intensive part of the algorithm and thus parallelizing it breeds immediate results for the betterment of the runtime of the program. In addition to this, we have also concluded that using quicksort for our sorting algorithm created a fairly significant bottleneck in runtime. This is because the worst case for quicksort is when something is already close to being sorted. As the program progressed, the fitness values being sorted became ever closer and thus cased significant slowdown while being sorted. Either utilizing another algorithm such as merge sort or parallelizing the existing quicksort would elevate this issue.

If we would continue development for this project, we will look to parallelize other portions of the program such as the mutate algorithm or the quicksort algorithm in ways explained above. One thing that we were not able to get to during this project, but expected to was to convert the letters from a grid of booleans to a bitmap image. Implementing this would also be beneficial. Training the algorithm on the entire alphabet so that it can differentiate between letters and random doodles would also be a useful extension of the project.