Genetic Algorithm Using CUDA

John Kotson and Kristen Morse
University of Southern California
kotson@usc.edu, kmorse@usc.edu

Introduction

Genetic algorithms can provide useful emergent solutions in systems that can be modeled as evolutionary; however, due to their time consuming nature, genetic algorithms are limited in what they can produce and how quickly they come to a satisfiable solution. Because the most fit member of each generation must be decided upon before the next generation can begin, a lot of the processing time is wasted during the evaluation phase of each generation. In order to get around this limitation, one can exploit the time saving properties of parallel programming by running the fitness function on each individual of a generation at the same time.

In our work, we use ideas behind parallel processing in an effort to reduce the time spent in the fitness function of our genetic algorithm. We develop our implementation for the CPU and then modify that code for use on the GPU in CUDA. We then proceed to evaluate our results and provide insight into our findings. Overall, we find that the parallelized version of our genetic algorithm runs faster on CUDA than on the CPU.

Approach/Implementation

Genetic algorithms follow basic evolutionary principles: natural selection, mutation, and crossover. Our genetic algorithm is modeled as any other. Each population is made up of individuals which are represented as a string of bits. A fitness function is calculated for each individual in the population and from that evaluation, a number of individuals with the best fitness functions are chosen as the elite group. As in natural selection, the group of elite individuals is left as is, while a crossover and mutation is performed on the rest of the population. For the general group, the "mutation" involves flipping a random number of bits within each individual. From this new generation, the process starts again from the beginning. This process is designed, as in evolution, to come closer to a target value each new generation. Eventually, the correct target should be reached.

It is important to note that while each generation is dependent on its predecessor generation, the calculations of the fitness function for each individual are independent of operations on other individuals. Consequently, the process of calculating the fitness of each member of a generation can easily be translated into a parallelized process. Furthermore, calculating each fitness function takes a large amount of time, so efficiently parallelizing the system is important to speedup. This becomes important when choosing which processor to run the algorithm on. As such, our team has chosen to develop code to run on the CPU and on the GPU in order to find the best fit.

Specifically, we developed code in C and ran it on the hpc-login3 cluster, modified the code, ran it on CUDA on the hpc-login3 cluster and compared results. Each “individual” was modeled as an equation of 32 bits (equations such as 4 + 5 etc). The strings of bits for the first generation were randomly generated for initialization. We gave the system a target value and the process would terminate once an equation or “individual” was equal to that value. At this point the fitness value would be zero. Smaller fitness values are preferred when natural selection takes place, such that the elite group is a number of individuals with the smallest fitness values. A threshold is set to limit the amount of generations created. If the fitness value does not reach zero by the time a threshold is passed (in our case, by the time the size of the population is equal to the number of generations), the program will terminate. The elite group remains the same while each member of the remaining group is mutated by flipping a random number of bits within the string. Over time this process reaches the target.

Analysis

Running our code on the hpc-login3 cluster yielded impressive results. Our CPU version ran in an average of 0.55908 seconds, while the CUDA version running on a GPU ran in an average of 0.25000 seconds. Overall, the code ran twice as fast on CUDA than on the CPU.

Implementing this theory in C code brought about some challenges. Because CUDA does not provide string support, modifying the code from the CPU version to the CUDA version was difficult and new string functions had to be written to allow the program to run the same. Additionally, we noticed performance bottlenecks during the selection process. While parallelizing the program helps reduce timing for calculating each member’s fitness, the natural selection cannot be made until all of these processes have terminated.

By running our genetic algorithm’s fitness function in parallel on CUDA, we were able to reduce the runtime of the script by almost half.

Conclusion

The nature of genetic algorithms allows them to be easily modified with parallel programming techniques. By harnessing the processing power of the GPU and exploiting parallel processing, an otherwise time consuming genetic algorithm is able to produce results in an acceptable amount of time. This opens up possibilities for improvement of many parallelizable programs.

Future Work

Using the conclusion from this paper, future researchers can be confident in running their genetic algorithms on CUDA and can expect a time speedup from parallelization on a CPU. This extends to other algorithms that can be parallelized as well, such as image processing.

Team Breakdown

Our team worked on this project together and pair programmed throughout the project. At first, John worked more on the CPU code. Kristen worked more on documentation and research. We both worked on the GPU CUDA code together.

References

Pospíchal, Petr, Jiri Jaros, and Josef Schwarz. "Parallel genetic algorithm on the cuda architecture." *Applications of Evolutionary Computation*. Springer Berlin Heidelberg, 2010. 442-451

Munawar, Asim, et al. "Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework." *Genetic Programming and Evolvable Machines* 10.4 (2009): 391-415.

Zhang, Sifa, and Zhenming He. "Implementation of parallel genetic algorithm based on cuda." *Advances in Computation and Intelligence*. Springer Berlin Heidelberg, 2009. 24-30.

Abderrahim A. (2011) Simple C genetic algorithm example [Computer program]. Available at http://pastebin.com/40Y5U1sY(Accessed November 2014)