Parallel Genetic Algorithms on the GPU

George Savin
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
georgesavin@cmail.carleton.ca

December 9, 2022

Abstract

Genetic Algorithms are very powerful metaheuristics that have been successfully applied in various disparate fields. While there are many sequential parts to the algorithm, a fair amount can and has been parallelized in MIMD and SIMD fashion. In this paper, we show a SIMT GPU solution to the the knapsack combinatorial problem. We do everything, including initialization of data on the device rather than the host CPU and show speed improvements across the board.

1 Introduction

Evolutionary Algorithms mimic behaviors of living things to search for optimal solutions. Genetic Algorithms are a well known subset of EAs, and employ genetic operators such as crossover, mutation and selection to ”evolve” new solutions in a solution space over many iterations (generations). Even with the main loop of GAs being domain independent, two main components end up being problem specific : encoding and evaluation. These components may dramatically increase execution time, especially when a lot of candidate solutions need to be evaluated every generation.

Thankfully, Genetic Algorithms are amenable to parallelization and indeed researchers have experimented with parallelizing everything from individual operators to the whole process. Different GA model schemes such as master-slave and island models have been parallelized, with more attention put on island models for their natural inclination towards distributed execution.

Empirical and experimental researchers have predominantly been the drivers of GAs, mainly as optimization tools, however designing parallel algorithms can differ quite fundamentally given the underlying architecture. Implementations on the GPU have gained in popularity as each solution can be acted on independently for parts of the algorithm. However, researchers [19], myself included, have in some fashion found the cost of learning these parallelization techinques in tandem with GPU technologies to be painful.

Nevertheless, many problems have been successfully solved with GAs such financial pattern discovery, MAX-3SAT, layout problems, and ML hyperparameter selection [16, 1] among many others.

To best explore this topic and our implementation with the reader, the paper starts with a quick subsection on particulars of CUDA GPU programming. Section 2 is a literature review of all the work done in this space up to now, while Section 3 is our problem statement as to why master-slave models need to be revisited in light of the more recent island model dominance. Section 4 contains our proposed solution for our CPU and GPU implementations. In Section 5 we describe our positive findings and how we improved upon our initial GPU implementation. Finally Section 6 contains our conclusions and future work.

1.1 CUDA GPU Programming

Programming on a GPU requires your code to be ran at times on a separate device. NVIDIA provides CUDA[1], a general purpose parallelization and programming model. We use the subsets on the C/C++ programming languge to run our GA on the GPU.

These NVIDIA GPUs consist of multiple streaming multiprocessors (SMs). When we define a kernel (the name of a function the host runs on the GPU), our blocks of threads get scheduled to run on these streaming multiprocessors. Each SM then executes these threads, with the lowest level of parallelism found at the *warp* level. Warps are groupings of 32 threads. Threads on the same block can synchronize and also access shared memory. Blocks however cannot synchronize, and do not have any shared memory access and must rely on slower global memory.

When it comes to memory access and best practices, we aim for warp level memory coalescing[2] of global memory, leading to less request transactions in tandem with memory access patterns such as interleaved addressing. We also need to avoid serialization of operations when attempting to retrieve data as the GPU distributes memory over banks.

Figure 1 highlights both the grid/block/thread structure as well as the different levels of memory available to our GPU programs.

2 Literature Review

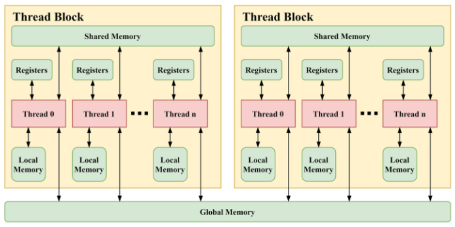
Using GPUs to accelerate Genetic Algorithms started to take off when NVIDIA released CUDA SDK 2.0 in 2008, allowing for more general programming tasks to be parallelized.

A first intuitive approach is to move an operator onto the GPU that can efficiently be ran in parallel, such as the fitness evaluation across a population [11]. Taking this idea further, generation of chromosomes could also moved to the GPU [4]. Unfortunately, this constant transfer between CPU and GPU at every generation slowed down the runtime, and depending on the population size, may not have even been worth it [14].

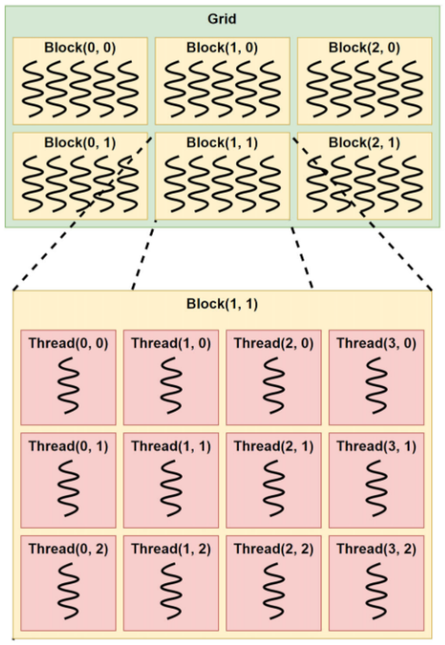
To overcome this transfer slowdown, master-slave models of a binary and real coded GAs were completely ported to the GPU [5, 2]. Each operator (tournament selection, two-point and single point crossover, bitwise XOR mutation) became separate CUDA kernels. Both the crossover and mutation kernels in these cases suffered from the possibility of having the same chromosome operated on many times, causing inefficient usage and propagation of sub-optimal solutions.

Because of the architectural and operational constraints of the GPU, such as limited shared memory and expensive global memory lookups, models that aimed to reduce global communication fared better. Inversely, these models seemed to have less overall accuracy [20], while other works found this difference non-existant [7]. Further follow-ups to quality of solutions between models have not been explored.

[1]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[2]https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html



(a) Memory Allocation Levels



(b) Thread, Block, Grid Distribution

Figure 1: CUDA GPU Layout of Threads and Memory

Island model GAs divide populations in the hope that diversity is preserved during evolution. On a GPU, this roughly translates to thread blocks as islands, and single threads to individuals. Within a block, fast memory access and synchronisation is available. Migration between islands occurs asynchronously. Early island model implementations [13, 19] used global memory only for migration. They were applied to numerical and combinatorial optimisations problems with tremendous speed-ups recorded. These speed-ups unfortunately were due mainly to comparisons with poorly optimized sequential GAs [9]. When compared against properly parallelized GAs on CPUs, the speed-up was more in line with expected theoretical analysis. A similar revision of speed-up was seen with newer master-slave implementations [17]. Trade-offs between speed and solution quality were analyzed and associated to parameter tuning, specifically island, generation and chromosome counts [18].

During this time, research looking at optimizing GA GPU representations as well as technique improvements to leverage more parallelization was taking off. Building on the island model implementations, simulated annealing was shown to provide faster convergence when replacing mutation [10]. Memory layouts were also explored, and chromosome based layouts proved to increase locality and make better usage of caches [7]. Different encoding representations also increased convergence speed at no detriment to solution quality [12].

Newer work incorporates some of these techniques, as well as developing new ones targeting Island models almost exclusively. Random seed improvement lead to a solution technique that only generates a single random seed, yet still benefits from the uniqueness and speed one would typically get by generating seeds every generation [18]. Another recent paper used the idea of synchronous migration intervals to improve solution quality by avoiding unintended migrations [8]. This one in particular also used the idea of allocating multiple threads per individual [15] combined with better data organization and found that their techniques provided up to 18x speedups and better solution quality compared to the original IMGAs on the same hardware. Newer work tries to use warp granularity to represent each island and reduce thread divergence [1]. They also perform synchronous and asynchronous replacement, improving solution quality, dubbing it Two-Replacement Policy. Unfortunately, these interesting island model changes were not compared to any prior island model implementations.

3 Problem Statement

Given the take-over of Island Models as the default model for genetic algorithms on the GPU, re-visiting the master-slave model last seen in [4] is a worthwhile endeavor. It gives us the chance to contrast the differences of these approaches on new GPU architecture and to apply the CUDA design methodology "Assess, Parallelize, Optimize, Deploy" (APOD) [3] while we improve our model. This re-exploration is the first we know of in almost a decade, and will show that the master-slave model is a worthy alternative, and arguably should be viewed as the default when first choosing to pursue a SIMT/SIMD acceleration of a GA.

We target the 01-binary knapsack problem as it has important applications in every day life, and proves to be a good benchmark.

4 Proposed Solution

Our CPU solution stays mostly faithful to the canonical Genetic Algorithm [6], and serves as a launching point for the APOD parallelization towards our GPU version. Figure 2 is a flowchart of our implemented algorithm. There are two GPU solutions as one represents a first parallelization attempt, and a second improved version, based on APOD changes leveraging NVIDIA GPU architecture. Differences between the two will be highlighted in each of the following operators.

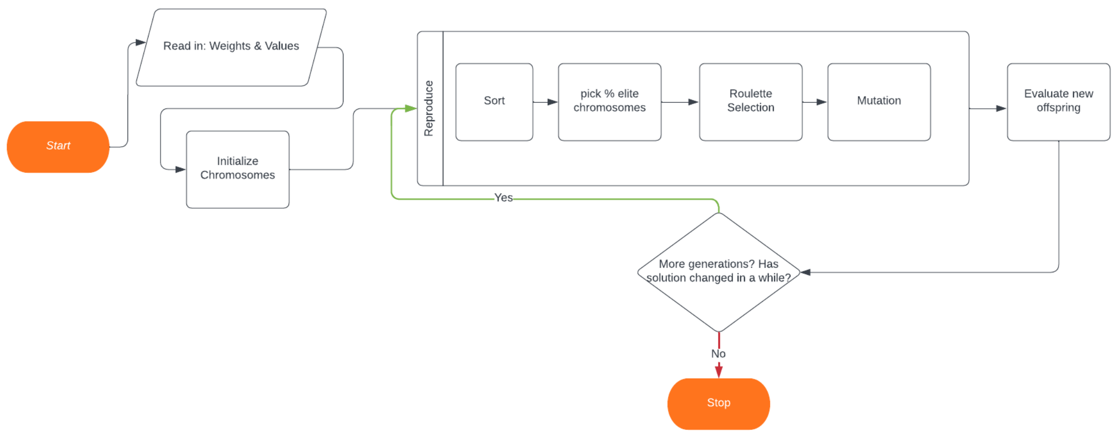


Figure 2: The Genetic Algorithm logical path for both the CPU and GPU version. The GPU version kernalizes these steps

4.1 Encoding Schema

A benefit of the binary knapsack is that both the genotype and phenotype representations are the same : a string of 1's and 0's indicating the presence or absence of the respective item at that index. For our problem, each 1 or 0 is represented by an integer, for a total array of length number of items. Figure 3 shows how the chromosome encoding lines up with the profit values and weight values of each item.

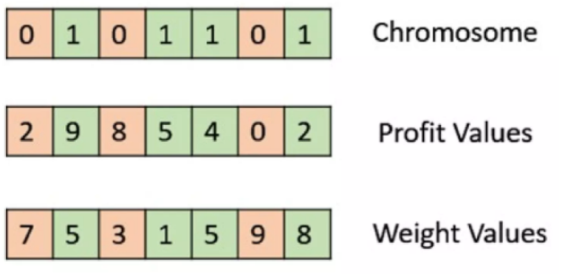


Figure 3: Binary Knapsack chromosome encoding

4.2 Initialization

There are multiple ways to initialize chromosomes, and when we have a lot of items, it might make more sense to use a greedy approach, or some a priori information about the values and weights. For our own knapsack problem however, on the CPU we decided to use a bernoulli distribution and a mersenne twister pseudo-random generator to determine if a gene was 0 or 1. We also ensured that the total weight was not greater than the knapsack capacity by setting the remaining genes to 0 if this capacity would be crossed by a chromosome.

The GPU solution had to be slightly modified as none of these were readily available, and we had to rely on the NVIDIA cuRAND[1] library which used the *XORWOW* generator. We also needed to split this step into two separate kernels *initKernel* which sets a cuRAND handle for each chromosome, and the actual initialize kernel, *initializeChromosomes* which sequentially iterated over genes and used a modulo to set either 0 or 1 values.

4.3 Evaluation

The CPU solution iterated over each chromosome sequentially and stored each respective score in the associated struct. A score of 1 was given to any chromosome that went over knapsack capacity. A running total, average and best score was kept throughout the loop. The GPU evaluation was done in parallel on each chromosome in the *evaluateChromosome* kernel, and the same penalty of lowering the total score to 1 was applied. Existing reduction algorithms on the GPU (in our case to sum the scores of chromosomes) only used arrays of floats or ints, and not over structs, so we needed an additional *pullScores* kernel to store this information into an array of scores that would then be reduced into a total sum by the *reduce* kernel. This *reduce* kernel was taken from the CUDA sample examples, and returned the summed result to the host.

[1]https://docs.nvidia.com/cuda/curand/index.html

A second improved GPU version leveraged a custom reduction we wrote and did not need to pull scores down from the chromosome structs. It also avoided moving the total score back to the host. This was all done in a kernel called *sumReducer*.

4.4 Reproduction

Reproduction is really an overarching function that relies on selection of parent chromosomes, actual cross-over/reproduction of these parents, and mutation of the resulting offspring. These steps need to be done sequentially, and this restriction applies to the GPU implementation as well. Afterwards, all the offspring are copied into the original chromosome array. The more optimized GPU implementation moves this copy into its own kernel that works at the gene level rather than the chromosome level to squeeze out more throughput.

4.4.1 Selection

Selecting parents were done by a roulette selection implementation that picked unique parents for crossover. All other papers reviewed overlooked this uniqueness in selection. Figure 4 visualizes this roulette concept. Spinning the wheel was simply the modulo between a random number and the total score sum. We spun the wheel twice for one parent, and the second time around, the wheel was spun with the first parent removed. On the GPU, this function lived on the device, and was called at the chromosome thread level. It could not be further parallelized due to the need to select a first parent.

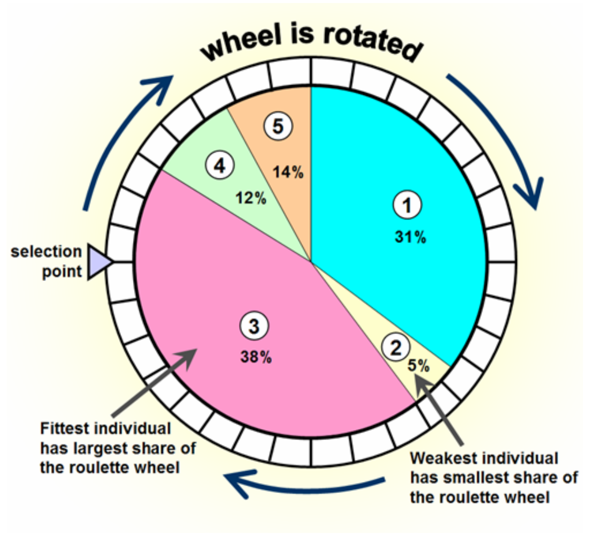
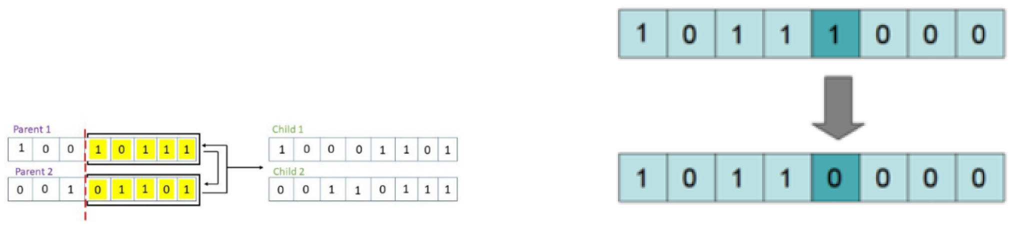

wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Figure 4: Roulette Selection of parent chromosomes

4.4.2 Crossover

The creation of offspring via crossover can be done in many ways, but for this project, we chose single point crossover. Figure 5a visualizes the single point crossover, where offspring are the parent chromosomes with swapped bits past the cutoff point. The CPU implementation allocates memory for the offspring, selects two parents and performs crossover. The GPU implementation is also at the chromosome thread level, and each thread performs this crossover, however only one of the offspring is chosen in a probabilistic manner. This was a change to ensure the algorithm would run on the GPU properly.



(a) Single-Point Crossover

(b) Gene Mutation

Figure 5: Single point Crossover and Mutation operations

4.4.3 Mutation

Finally mutation is when a gene flips from 1 to 0, or 0 to 1. It usually has a very small probability of occurring, and in our case, we kept it at 0.001. The first GPU implementation does this in parallel at the chromosome level, whereas the second implementation does it at the thread gene level increasing throughput.

4.5 Experiments

Knapsack item lists of various sizes were used, along with varying population sizes to debug and test both CPU and GPU programs, however the final experiments were for a 50 item knapsack and a 1000 item knapsack. With these two sizes, population sizes ranging from 32 to 1024 were tested against the CPU, 1st GPU implementation and post APOD 2nd GPU implementation. NIVIDIA Nsight systems[3] was used to profile each GPU run.

5 Experimental Evaluation

5.1 Hardware

The CPU execution was done on an Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz, with 32GB of RAM. The GPU used was a 12GB RTX 3060 on a Carleton cloud VM.

5.2 Results

In Figure 6, we can see at first glance that the second optimized GPU implementation is faster than the first one, and in fact, as population increases, also has a smaller gradient than the original GPU implementation. These linear increases are in contrast to the exponential rise of the CPU version, and indeed, both GPU solutions perform better than the CPU one. Near 1000 population size, the 2nd GPU implementation was 2.8x faster than the 1st GPU one, and 5.2x that of the CPU.

Shared memory was only leveraged in the reductions, and we had to rely on global memory to avoid serialization and some cache misses. We also had to rely on global memory due to the size of our arrays and the limited space on the blocks and threads. Table 1 goes

[3]https://developer.nvidia.com/nsight-systems

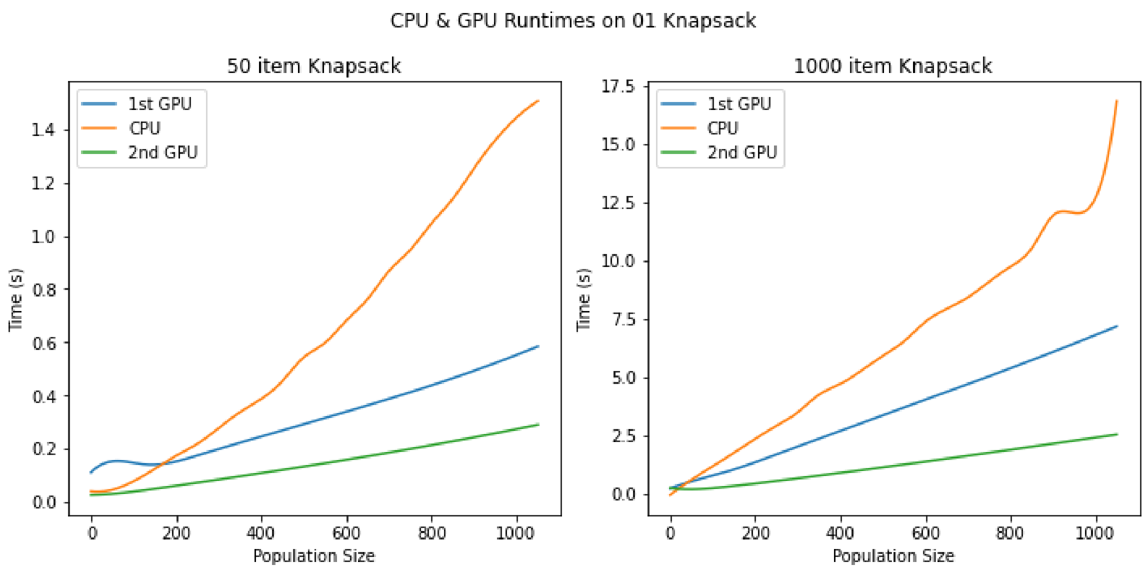


Figure 6: Results of the GA for our implementations across the GPU and CPU

over the APOD changes that ended up working in our favor while improving the first GPU algorithm. The Reproduction kernel took up 91% of the all kernel execution time, so our optimizations aimed at reducing this number as well as the total time. Table 1b shows the halving effect of pulling out mutation from a device function to its own global kernel where each threads main job was to determine if a bit needed to be flipped. This concept was also extended as seen in Table 1c to the copying of offspring back into the parent chromosomes for the next generation, and this also led to another halving of the total reproduction function runtime. Taking these out did produce some additional kernel call overhead, but it was well worth it in our case.

Naively, we assumed that it would then be possible to run all kernels and threads at the gene level, but we quickly realized there was a lot more thread divergence and less memory coalescence. In addition, complications occur when chromosomes are spread across two separate CUDA blocks. We can no longer do reductions across these blocks without using atomics, and we also lose the ability to sync between threads in case we want to continue with the sequential nature of our algorithm. We attempted to work around this by using atomicity specifically for the threads of a chromosome that spanned two blocks, but this caused too much divergence and the runtime was not any better than what we had achieved in the 2nd GPU implementation.

We tried various other memory representations in 2D to try and get around this limitation, but none of the runtimes improved in any significant way.

6 Conclusions

Not only is acceleration of Genetic Algorithms via GPUs faster than CPU implementations, a master-slave model is not too much technical overhead and will provide good speed-ups even in less than ideal implementations.

The remainder of the time is then spent on tweaking the memory layout and actions done in threads to increase cache hits and reducing thread/warp divergence as much as

Time (%)	Total Time (ns)	Instances	Avg (ns)	Name of Kernel
91.0	12,396,614,221	5,994	2,068,170.5	GPUreproduceChromosomes
8.7	1,183,193,740	6,000	197,199.0	evaluateChromosomes
0.1	16,970,440	6,000	2,828.4	sumReducer
0.1	14,235,215	6	2,372,535.8	initializeChromosomes
0.1	12,529,457	6,000	2,088.2	pullScores
0.0	1,758,771	6	293,128.5	initKernel

(a) First APOD pass, using the custom sumReducer kernel. This reduced memory slowdown moving from device to host back to device

Time (%)	Total Time (ns)	Instances	Avg (ns)	Name of Kernel
83.5	6,782,679,608	5,994	1,131,578.2	GPUreproduceChromosomes
14.9	1,209,217,233	6,000	201,536.2	evaluateChromosomes
1.0	85,121,386	5,994	14,201.1	mutateChromosomes
0.2	16,672,049	6,000	2,778.7	sumReducer
0.2	13,412,805	6	2,235,467.5	initializeChromosomes
0.2	12,219,734	6,000	2,036.6	pullScores
0.0	1,705,907	6	284,317.8	initKernel

(b) We pull out the mutateChromosome function into its own kernel that now acts on the gene level. This added throughput halves the runtime of the reproduce kernel while our new kernel's runtime is negligible

Time (%)	Total Time (ns)	Instances	Avg (ns)	Name of Kernel
70.0	3,313,793,979	5,994	552,851.8	GPUreproduceChromosomes
25.5	1,209,338,925	6,000	201,556.5	evaluateChromosomes
1.8	85,082,093	5,994	14,194.5	mutateChromosomes
1.7	82,532,905	5,994	13,769.3	copyOffspringIntoChromosomes
0.4	17,049,045	6,000	2,841.5	sumReducer
0.3	14,001,103	6	2,333,517.2	initializeChromosomes
0.3	12,340,733	6,000	2,056.8	pullScores
0.0	1,717,651	6	286,275.2	initKernel

(c) The copying of offspring back into parent chromosomes can also be done at the gene level for maximal throughput. We can see doing this again halves the runtime of the Reproduce kernel, with negligible runtime for our new copyOffspring kernel

Table 1: APOD Improvement via NVIDIA Nsight profiling the 1000 item knapsack runs across 6 (32,64,128,256,512,1024) population sizes

possible. This not only takes away from the actual problem being solved, but is highly tied to the actual GPU hardware available. NVIDIA has shown no qualms with changing architecture, and very recently announced a new paradigm of shared memory across blocks, which is something that we were not able to leverage in our project. Indeed, a GPU solution on a non NVIDIA GPU may look very different, and due to the painful learning process mentioned in the introduction, not many have ventured into this even less documented space.

Sometimes, as we saw in this project, threads working on every gene at once is an obvious and ideal speed-up, but one that does not always work given the encoding and problem at hand. Island Models in contrast try to avoid this by having only as many chromosomes as they can fit in their definition of an island (usually a block [8], but people have tried this at the warp level as well [1]). Unfortunately, the smaller amount of chromosomes per island comes at a cost of overall solution quality.

6.1 Contributions

We showed that not only is GPU acceleration of GAs do-able, but the direct master-slave approach with some architectural provisions is less technical overhead than island models and provides tempting speed-ups.

We are also the first to walk through systematic changes between gene and chromosome level thread execution in our implementations, and show profiling results to support our decisions.

6.2 Future Ideas

We could not find any implementations of the Island GPU papers, so attempting to implement them and compare with our results would be interesting. Given the small and decreasing amount of chromosomes per island as the encodings get larger, we forsee less successful runs. However, newer variations of Island Models with the upcoming block group shared memory and synchronization would be worthy of exploration as there would be a new division of an island possible on GPUs.

No fine-tuning of grid and block sizes were done, so an extension looking at these and other possible parameters on the GPU would be good to get further speed-ups.

We also did not have time to explore more dense encodings of our chromosomes, which could have been done at the bit level and stuffed into integers. This simple change would for sure speed up many kernels as masks could then be used.

Finally, more intricate Genetic Algorithm idea's could be implemented, and a focus more on quality of score rather than just speed would be the differentiating factor between algorithms.

References

[1] Faiza Amin and Jinlong Li. Two-Replacements policy island model on GPU. In *Advances in Swarm Intelligence*, pages 242–253. Springer International Publishing, 2022.

[2] Ramnik Arora, Rupesh Tulshyan, and Kalyanmoy Deb. Parallelization of binary and real-coded genetic algorithms on GPU using CUDA. In *IEEE Congress on Evolutionary Computation*, pages 1–8, July 2010.

[3] Thomas Bradley. Assess, parallelize, optimize, deploy, Jul 2012.

[4] Stefano Cavuoti, Mauro Garofalo, Massimo Brescia, Antonio Pescape', Giuseppe Longo, and Giorgio Ventre. Genetic algorithm modeling with GPU parallel computing technology. In Bruno Apolloni, Simone Bassis, Anna Esposito, and Francesco Carlo Morabito, editors, *Neural Nets and Surroundings: 22nd Italian Workshop on Neural Nets, WIRN 2012, May 17-19, Vietri sul Mare, Salerno, Italy*, pages 29–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[5] Debattisti, Marlat, Mussi, and others. Implementation of a simple genetic algorithm within the cuda architecture. *The Genetic and*, 2009.

[6] John H Holland. Genetic algorithms and adaptation. In *Adaptive control of ill-defined systems*, pages 317–333. Springer, 1984.

[7] Paul Jähne. Overview of the current state of research on parallelisation of evolutionary algorithms on graphic cards. *Informatik 2016*, 2016.

[8] Dylan M Janssen, Wayne Pullan, and Alan Wee-Chung Liew. Graphics processing unit acceleration of the island model genetic algorithm using the CUDA programming platform. *Concurr. Comput.*, 34(2), January 2022.

[9] Jiri Jaros and Petr Pospichal. A fair comparison of modern CPUs and GPUs running the genetic algorithm under the knapsack benchmark. In *Applications of Evolutionary Computation*, pages 426–435. Springer Berlin Heidelberg, 2012.

[10] Cheng-Chieh Li, Chu-Hsing Lin, and Jung-Chun Liu. Parallel genetic algorithms on the graphics processing units using island model and simulated annealing. *Advances in Mechanical Engineering*, 9(7):1687814017707413, 2017.

[11] Ogier Maitre, Laurent A Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1403–1410, New York, NY, USA, July 2009. Association for Computing Machinery.

[12] Martín Pedemonte, Enrique Alba, and Francisco Luna. Bitwise operations for GPU implementation of genetic algorithms. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 439–446, New York, NY, USA, July 2011. Association for Computing Machinery.

[13] Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the CUDA architecture. In *Applications of Evolutionary Computation*, pages 442–451. Springer Berlin Heidelberg, 2010.

[14] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Population parallel gp on the g80 gpu. In *European Conference on Genetic Programming*, pages 98–109. Springer, 2008.

[15] Rajvi Shah, P Narayanan, and Kishore Kothapalli. Gpu-accelerated genetic algorithms. *cvit. iiit. ac. in*, 2010.

[16] Prakruthi Shivram. *Parallelization of Genetic Algorithm to Solve MAX-3SAT Problem on GPUs*. University of South Florida, 2019.

[17] Rashmi Sharan Sinha, Satvir Singh, Sarabjeet Singh, and Vijay Kumar Banga. Speedup genetic algorithm using C-CUDA. In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 1355–1359, April 2015.

[18] Xue Sun, Ping Chou, Chao-Chin Wu, and Liang-Rui Chen. Quality-Oriented study on mapping island model genetic algorithm onto CUDA GPU. *Symmetry*, 11(3):318, March 2019.

[19] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based island model for evolutionary algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1089–1096, New York, NY, USA, July 2010. Association for Computing Machinery.

[20] Long Zheng, Yanchao Lu, Mengwei Ding, Yao Shen, Minyi Guoz, and Song Guo. Architecture-based performance evaluation of genetic algorithms on Multi/Many-core systems. In *2011 14th IEEE International Conference on Computational Science and Engineering*, pages 321–334, August 2011.