X-Aevol: GPU Implementation of an Evolutionary Experimentation Simulator

Laurent Turpin
laurent.turpin@inria.fr
Univ. Lyon, Inria, CNRS, ENS de Lyon, UCBL, LIP
Lyon, France

Thierry Gautier
thierry.gautier@inrialpes.fr
Univ. Lyon, Inria, CNRS, ENS de Lyon, UCBL, LIP
Caluire et Cuire, France

Jonathan Rouzaud-Cornabas
jonathan.rouzaud-cornabas@inria.fr
Inria Grenoble Rhône-Alpes, France
Univ. Lyon, INSA Lyon, Inria, CNRS, LIRIS, France
Lyon, France

ABSTRACT

X-Aevol is the GPU port of the Aevol model, a bio-inspired genetic algorithm designed to study the evolution of micro-organisms and its effects on their genome structure. This model is used for *in-silico* experimental evolution that requires the computation of populations of thousands of individuals during tens of millions of generations. As the model is extended with new features and experiments are conducted with larger populations, computational time becomes prohibitive.

X-Aevol is a response to the need of more computational power. It was designed to leverage the massive parallelization capabilities of GPU. As Aevol exposes an irregular and dynamic computational pattern, it was not a straightforward process to adapt it for massively parallel architectures. In this paper, we present how we have adapted the Aevol underlying algorithms to GPU architectures. We implement our new algorithms with CUDA programming language and test them on a representative benchmark of Aevol workloads. To conclude, we present our performance evaluation on NVIDIA Tesla V100 and A100. We show how we reach a speed-up of 1,000 over a sequential execution on a CPU and the speed-up gain up to 50% from using the newer Ampere micro-architecture in comparison with Volta one.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; *Artificial life*; *Genetic algorithms.*

KEYWORDS

Genetic Algorithm, *in-silico* Experimentation, GPU, Parallelization

ACM Reference Format:
Laurent Turpin, Thierry Gautier, and Jonathan Rouzaud-Cornabas. 2021. X-Aevol: GPU Implementation of an Evolutionary Experimentation Simulator. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463195

1 INTRODUCTION

In biology, experimental evolution consists in studying the effect of evolution on living organisms thanks to observations. As evolution is a long term process, experiments have to be of long term like the famous LTEE lead by Richard Lenski [17]. This 33-year ongoing experiment has shown very interesting results [18] but it is a very long, time consuming experiment. Moreover, having an experiment starting from the beginning of life is impossible as life arrived more than 3 billion years ago without a user manual.

As many computational models helped understand real life thanks to simulations [6, 16], several *in-silico* experimental evolution software have been developed with different models [10, 11, 22] to study evolution via computer simulations. An example of such model is Aevol[1] [13]. All these models simulate a population of individuals with a set of genetic materials that evolve through a process of selection, reproduction and mutation. In fact, all these models are genetic algorithms. However, an important difference has to be emphasised. Classical genetic algorithms [1, 12, 24, 25] are designed to solve a given useful problem. For Aevol (and other *in-silico* experimental evolution software), the final result is not important. The purpose is to study the evolution process (*i.e.* the paths within the fitness landscape) and its emerging mechanisms.

Nevertheless, in both cases, performance is needed from these algorithms, either to quickly have solutions or, for experimental evolution software, to quickly have results. For the latter, the implemented models can be complex with several parameters that influence the evolution. Experimental campaign can then execute hundreds of simulations with parametric exploration and repetitions to gain statistical power to test hypothesis such as shown in [19].

This is where the HPC is required to allow such experiment to finish in reasonable time. To accelerate programs computations, algorithms can be parallelized on multi-CPUs architectures and even on clusters with multiple nodes. However, for some years now, GPUs are offering massive parallelism capabilities with large data bandwidth for a better energetic and economic efficiency. The use of GPUs are unfortunately far from being straightforward because, in order to benefit of this massive parallelism, software have to expose this parallelism and algorithm must be redesigned.

In the case of Aevol, the model leads to a memory bounded algorithm using highly irregular data structures including nested data with unknown size. Parallelization of such irregularities is not an easy task. Therefore, our contribution is X-Aevol, a port of the

[1] http://aevol.fr/

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463195

Aevol model on NVIDIA GPU using CUDA. X-Aevol is a prototype that focuses on the fitness evaluation of the individuals as it is the most time consuming part of the software and this is there that lie the algorithmic complexity of the model. In this paper, we present our new algorithms allowing to express more parallelism. We compare X-Aevol with a CPU implementation of Aevol and show that X-Aevol algorithms are leading to a speedup of up to 1,000 against sequential CPU executions.

For the remainder of the paper, the section 2 presents the genetic algorithm of Aevol. Then, the section 3 describes the implementation of the GPU parallelization which is evaluated in the section 4 thanks to a representative benchmark of Aevol workloads with different GPUs. Finally, the section 5 discusses related works and we conclude and present future work on X-Aevol in the section 6.

2 AEVOL

Aevol implements a bio-inspired genetic algorithm. It depicts a population of individuals defined by their genome. As shown in Figure 1, the genome (a) is composed of one circular double stranded chain of *bases* (or base pairs because of the double strand feature). Unlike real life DNA, the bases are not the usual A, C, T or G but they are 0 and 1. It is a binary DNA where 0 pairs with 1 and *vice versa.*

The population is spread on a 2D toric grid whose size is a parameter of the model. In each cell lives one and only one individual. During the evolution, a local selection is done for each cell where the 9 neighbors compete for reproduction. After reproduction, mutations may happen on individuals' genome depending on the mutation rate, another model parameter. The mutations can take the form of point mutations (switching one base or inserting or deleting a small number of base pairs) or large rearrangements (deletion, duplication, inversion and translocation). Consequently, during the evolution, the genome size of the individuals will change and among a single population, individuals will have different genome sizes. This size can go from a few hundreds (and fewer) to hundreds of thousands (and larger) and brings lots of irregularities in the genome structure.

Before repeating the evolution cycle, we have to evaluate the individuals to determine their fitness. In Aevol, the genome of an individual has to pass through a bio-inspired process to get the fitness of the individual. More precisely the genome is treated to first get the phenotype of the individual and then, the phenotype is compared to a target which is a parameter of the model. The distance to the target corresponds to the fitness.

The next subsection will explain the details of the evaluation.

2.1 Evaluation process

The Aevol model wants to mimic the genome of a bacteria with a complex evaluation process. The genetic material passes through a phase of transcription, translation and folding to eventually get to the phenotype that takes the form of a function p. Knowing the target t, also in the form of a function, the more p fits t, the greater will be the fitness of the individual. The sections 2.1.1 to 2.1.3 will explain all these phases as shown in Figure 1.

2.1.1 *Transcription or Finding RNAs (b) and (c).*

The transcription is the action of creating all the RNAs from the genome. RNAs correspond to segments inside the DNA delimited by a promoter followed by a terminator. The pattern of a promoter is a 22 bases sequence that allow up to 4 errors. The number of errors will determine the expression level e of the RNA. A terminator is not a particular pattern but rather a rule to follow exactly ($abcd * * * \neg d \neg c \neg b \neg a$ with a, b, c and d booleans).

Two interesting points can be noticed: 1. one terminator can end multiple RNAs if multiple promoters stand before it, resulting in overlapping RNAs and 2. all the bases outside RNAs will not participate to the phenotype and are considered non-coding.

2.1.2 *Translation, or Finding Genes (d), (e) and (f).*

Once the locations of all the RNAs are found, genes can be found and translated into proteins. Genes, as RNAs for the DNA, are segments inside a RNA started by a Shine-Dalgarno sequence with a START codon and ended by a STOP codon. Within the translation process, the sequence is read 3 bases by 3 bases. In fact, 3 bases of a gene correspond to one codon (translating to a single Amino Acid). It means that the STOP codon must be at a distance multiple of three from the START. Moreover, the STOP codon cannot stand beyond the end of the RNA terminator. If no STOP is found for a START, the gene cannot be translated into a protein. Finally, as RNAs, genes can overlap.

To translate the gene into a protein, the gene is read codon by codon. Within our model, it exists $2^3 = 8$ different codons (d). 2 of them are reserved for START and STOP and the 6 others can be composed to create and define a protein (e). In Aevol, a protein is defined by 3 parameters: m include in $[0, 1]$, h include in $[-1, 1]$ and w include in $[0, max\_width]$ (max_width is one of the program parameters). These 3 values define a triangle as shown in (f).

2.1.3 *Folding or Compute the Phenotype and Fitness (g).*

Once the set of all proteins is computed, the phenotype is then defined as the sum of all the proteins' triangle. The phenotype is finally compared to the target to get the fitness of the individual as a scalar number.

2.2 The computational problem

The Aevol model, as described previously, is design to study the impact of evolution on the structure of the genome, its size, the density and distribution of genes or non-coding bases, and what phenomena are in action. Input parameters, like the mutation rates or the population size, can have effect on how these phenomena occur during the evolution. Experiments may need to explore multiple combinations of these parameters during millions of generations and with multiple repetitions to confirm or reject a hypothesis. In order to have results in a reasonable delay, the Aevol software needs to be as fast as possible. More precisely the computation of one generation must go as fast as possible as the evolution process is inherently sequential (generation by generation). However, for now, experiments are limited in population size (no more than 4,096) and adding complexity the model is a risk of having too long experiments.

A parallelization of Aevol has been made on multi-CPU architectures [26]. It shows that the random mutations lead to irregularities on the computation time of individuals, mostly because of the large

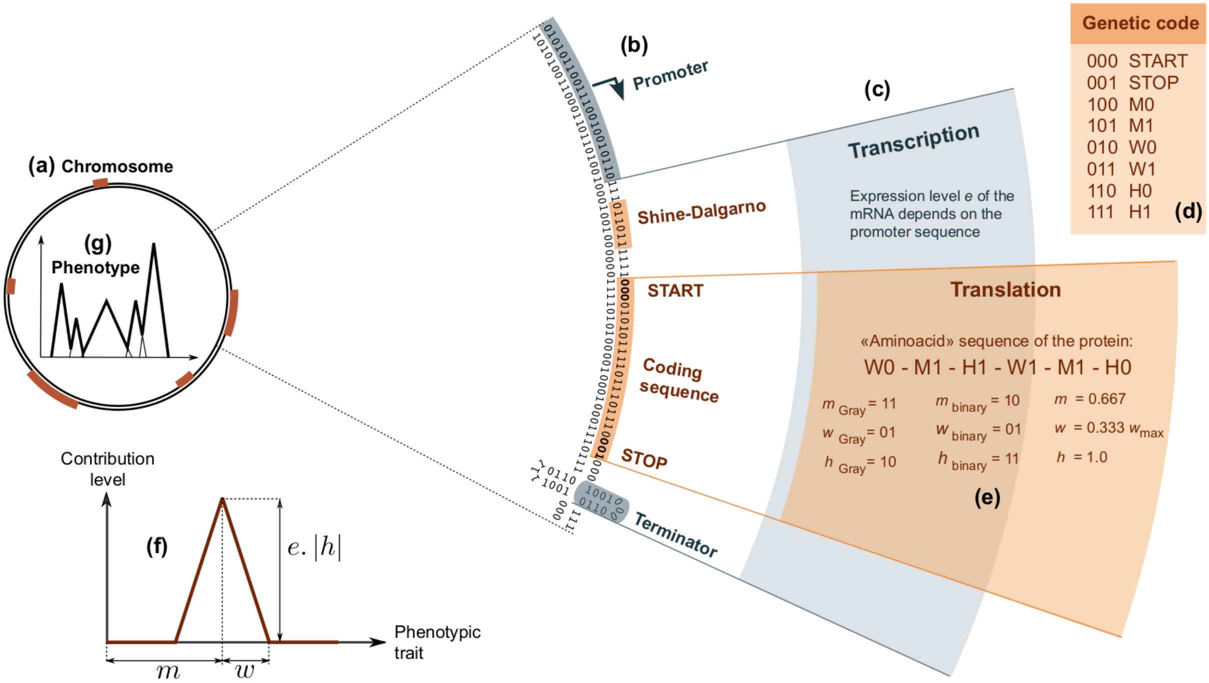

Figure 1: Illustration of the evaluation process of Aevol from the genome to the phenotype. Figure taken with authorization from [15]

distribution of genome size, and show an interesting scheduling problem. The paper also showed that this parallelization is limited and that improvements are possible if more parallelism is expressed by the program. For that end, GPUs are required as they can handle much more parallelism than multi-CPUs. Therefore, our goal is to implement X-Aevol, the GPU port of Aevol, to take advantage of the computational power of GPUs. X-Aevol will first be implementing the evaluation process of the individuals as this is the most time consuming part of the simulation [26] and present most of the algorithmic complexity.

3 FROM CPU TO GPU

This section explains how we have redesigned the Aevol algorithm to fit a massively parallel architecture such as the one exposed by GPU. It shows what are expected from GPUs and how algorithms must be adapted to benefit from their computational power. These algorithms have been implemented for NVIDIA GPUs using CUDA.

3.1 What is GPU Parallelization?

A GPU is a massively parallel processing unit designed to handle lots of similar computations at once. First used for graphics purpose, computer scientists have quickly used GPUs for general-purpose computations [23]. From the hardware perspective, a GPU is composed of many Streaming Multiprocessors (SMs) which contain many cores that execute instructions. Next to this computational hierarchy stands a memory hierarchy composed by a global memory accessible from any SM helped by a faster L2 cache, and in each SM stands an even faster L1 cache, a Texture memory and registers.

From the software point of view, the programming model of a GPU asks the developer to execute GPU code, called kernel, by a grid of blocks which each contains a grid of threads. Blocks should be considered as independent from each other and can be executed in any order. All the threads of a kernel will follow the same base of code but with different data, hence the SIMT (Single Instruction Multiple Threads) programming model. However, they can still diverge from each others thanks to conditional branching. Threads from different blocks have no problem with branching as blocks are independent. On the other hand, branching inside a block can reduce efficiency because of the software/hardware limits. From a memory point of view, NVIDIA GPUs expose 3 types of memories: 1) a very small, high bandwidth and low latency that is specific to each thread, 2) a medium size, high bandwidth and medium latency that is shared among threads of a block, 3) a large size, medium bandwidth and latency that is accessible from any blocks running on the same GPU. All memory levels expose synchronization primitives. Nonetheless, the cost of synchronization is increasing as we go up into the memory hierarchy.

To map the software to the hardware, blocks are scheduled on SMs and the shared memory lies on the L1 cache. A SM can handle multiple blocks by interleaving their execution. Inside a SM, the cores execute warps that correspond to groups of threads. The key principle is that every thread of a warp executes the same instructions at the same time. To allow branching, some threads on a warp can be deactivated but it means that efficiency is lost if there is too much divergence (*i.e.*, some cores are idle). It is why branching should be avoid inside a block. Warps are scheduled on cores and it is beneficial to have multiple warps (from one or multiple blocks) on one SM because while some warps wait for data to be loaded, others can be executed on cores. In actual implementations, warps contains 32 threads and block should contains multiple of 32 threads.

To give an analogy with traditional CPUs, SMs can be compared with CPU cores and GPU cores can be compared to vector units of CPU cores. In a sense, this two-level parallelization already exists on CPU but with less efficiency and more programming complexity. A GPU is designed to support this massive parallelization and offers easier programming model. However, developers have to express as much parallelization as possible inside their code to use correctly the computational power of a GPU. Algorithms should be changed to fit this new paradigm.

In a software, the kernels are more often launched by the CPU *host* with the CUDA API on the GPU *device*. The size of block grid and thread grid is chosen at the kernel launch. The kernels can be launched asynchronously and the device will put them in a queue to execute them in order. The series of kernels will then be executed in order, one after the other. Upon that, the CUDA host API offers possibilities to have multiple queues executing concurrently on the device and even express a graph of tasks that will use data dependant synchronizations. About the memory, the host memory (*i.e.*, from the CPU) is separated from the device memory (*i.e.*, from the GPU). If data must be accessed on the device and on the host, data transfer between them must occur. These transfers can be a real bottleneck for a software that has a part of its workload executed on the host and the other part on the GPU. Indeed, these transfers pass through bus (PCIExpress) that have larger latency and lower bandwidth that the one of the device or the host. In the worst case, memory transfer can dominate execution time and using GPUs could lead to a loss of performance. Fortunately, data transfer can be overlapped with computation.

Using all this information, the next section will present the porting of the Aevol model on NVIDIA GPU.

3.2 Porting Aevol on NVIDIA GPUs

The GPU implementation of the evaluation process of Aevol has been made using the CUDA language and API. To code kernels, the CUDA language includes many features of the C++ language (classes and structs, lambda expression, `new` and `delete`, ...) but it does not include the C++ STL and especially the data structures (vectors, lists, ...). CUDA programmers can use the Thrust interface to benefit some structures and algorithms from C++ STL, however it is mostly a host side interface and does not provide a powerful enough memory abstraction to use the GPU power entirely[8]. Arrays must be managed in the C old way with manual memory allocation. These allocations can be done from the host, however the size of all the arrays must be known in advance. Allocations can also be done from a kernel with a `malloc` primitive to have more dynamic allocations. Unfortunately, even if `malloc` is available, the performance of these dynamic allocations are very poor and despite the fact that solutions are proposed by the community [7], the CUDA library does not integrate these solutions. Consequently, dynamic allocation should be kept as low as possible.

As the evaluation of an individual is independent from the one of others, it was quite straight forward to allocate one block per individual. It means that for each step of the individual's evaluation (see section 2), we have to find a way to parallelize the problem to use the available threads. Some of the algorithms were simple to adapt (*e.g.*, the computation of the fitness can be seen as a set of vector operations) but other were much less obvious. In the remaining of this section, we present 2 kernels that have required new design compared to CPU implementation.

3.2.1 *Searching patterns and recording their locations.*

For the *Search Patterns* kernel, we can parallelize over each base pair. As described in the listing 1, each thread handles a base pair, check for all patterns, and go to its next base pair using a stride mechanism. Indeed, as the number of threads inside block is bounded and the Aevol genomes are not, striding is necessary. The algorithmic complexity is of $O(size_{genome} \times size_{pattern})$ but with a great potential of parallelization. From this point, another problem arises: how do we record the patterns locations?

```
int idx = threadIdx.x;
int stride = blockDim.x; // nb_threads per block
for (int pos = idx; pos < size; pos += stride)
    check_pattern(pos);
```

Listing 1: Searching pattern algorithm

On a sequential CPU, as soon as a pattern is found, you push back the location (*i.e.*, the position within the genome) to a vector that will automatically allocate more memory if needed (with overheads). On a GPU, doing the same will lead to a large overhead. Indeed as previously stated, memory allocation from kernel is to avoid due to high performance cost. Moreover, due to the large number of threads that are working at the same time, concurrent access to the stored data will happen.

Consequently, we change from a dynamic to a static allocation thus avoiding high overhead `malloc` calls. Nonetheless, how to predict the required memory for each array? In practice, we allocate enough memory (and certainly too much) from the host side respecting an upper bound. For example, there will never be more promoters than the genome size. For the concurrent accesses, a first idea was to use an array of booleans that has the same size as the genome. For each position, if a pattern is at this position, a `true` value is indicated and if there is no pattern, a `false` value is indicated. By doing so, you need one array of booleans for each pattern. In that way, setting the boolean array can be easily done in parallel. The last problem is to compact the array of booleans to only keep the locations. The problem is known as Stream Compaction [3].

With the first version of our algorithm (`seq_SC`), the compaction is done sequentially, *i.e.*, one thread does all the work. This is an easy solution but parallelization is impossible.

Consequently, we have looked at algorithms allowing to parallelize Stream Compaction problems. It turns out that there is two possible solutions to do that. You can use prefix sums with a parallel scan algorithm [2, 21], or you can just use atomic operations. In our case, we realize that patterns were not so frequent inside genomes and atomic operations were possible without having all the threads waiting on each others to push back a location resulting in a sequential execution. Moreover, by using atomic operations, the array of booleans turns out to be useless and the algorithm (`par_SC`) looks a lot like the sequential algorithm but with atomic operations.

Our algorithm to search patterns is used to find the promoters, the terminators and the gene-starts on both strand. Promoters aside, it is of interest if the terminators and gene-starts arrays are sorted to quickly search in them [2]. There is no guarantee that these arrays are sorted with the method of atomic operation. Nothing guarantees that a thread searching at an earlier position will push back its position before a thread at a latter position. That is especially true with multiple warps for one block because there is no guarantee on the scheduling order between warps. With the case of a single warp in a block though, we know that each stride will execute in order. It turns out that, after many experiments, when using one warp per block for pattern searching, arrays were sorted after the execution of the kernel. We suppose, with no confirmation, that atomic operations inside a single warp could be treated in order of thread id. Unfortunately, this is an unspecified behaviour that is not safe for production software.

3.2.2 Processing metadata: Transcription and Translation. The arrays of patterns are called metadata. After filling all the metadata, transcription is done, it means that for each promoter we search for the closest terminator on the same strand. Then for each RNA, we have to find all the gene-starts inside the boundaries. For translation, we choose to see each gene-start inside RNAs as potential genes. From this position, we translate triplets of bases into codons until finding a STOP one. If we go beyond the RNA limit, the gene is considered as non functional. Thus, we have read codons for nothing.

By following such an algorithm, we can expose a level of parallelism that correspond to the number of RNAs (or promoters). The GPU port is quite straightforward: a thread is allocated for each RNA. For the translation process, we can expose a level of parallelism that corresponds to the number of genes. But as described in section 2, the model exhibits a nested structure of data *i.e.*, there is a list of genes for each RNA. We could allocate multiple threads for each RNA to eventually have one thread per gene. However, as the number of genes inside a RNA is not known in advance and this irregularity come from program parameters, we would have wasted a large number of threads. Accordingly, a more efficient way to do so is to extract genes from the RNAs to know their number and have direct access on them. By doing so, the problem becomes similar to the previous Stream Compaction and can be solved by similar ways.

Our first solution (`seq_scan`) was to browse all the RNAs sequentially to extract genes inside a non nested array. The lack of parallelism of this solution made us think of a more parallel solution. As Stream Compaction, we could use atomic operations or prefix sums with a parallel scan algorithm. This time, we expect to find genes inside RNA and atomic operations will certainly sequentialize the process. Therefore, we implemented (`par_scan`) a simple scan algorithm [21] to solve this problem. Thanks to this prefix sum, instead of one thread to fill the non nested array, we can use at most as many threads as RNAs. In practice we used 32 threads with a striding mechanism. The process of finding genes and record them inside the array is done by the *Find Genes* kernel. After this kernel, translation can be done with more parallelism.

One could notice the problem of allocating memory for the non nested gene array whose size is unknown. Unlike metadata, we did not allocate in advance memory from the host. Instead, we used `malloc` inside kernel but with a custom implementation of a simple Vector that will reallocate memory only if needed. We hope (and know from many experiments with Aevol on CPU) that through multiple generations, the number of genes is not going to change too much and that only few generations will suffer reallocation while most will just reuse the already allocated memory. As mentioned earlier, dynamic memory allocation, although very convenient, is to avoid on GPU for now [7]. Some experiments show that execution of the *Find Genes* kernel is a dozens of time longer if allocation is done.

The GPU port of the rest of the evaluation process exposes less algorithmic and performance issues and will not be detailed. With the evaluation model completely implemented on GPU, we can evaluate the different solutions.

4 EVALUATION OF THE X-AEVOL PROTOTYPE

In this section, we present the performance evaluation of our port of the Aevol evaluation process on GPU. We first show and explain the performance of our algorithms that tackle the two main computational problems encountered during the development. Then we discuss the performance of the entire evaluation process on GPU. We measure performance on two different GPU micro-architectures: the NVIDIA V100 with 80 SMs and the NVIDIA A100 with 108 SMs. As inputs, because our implementation cannot make artificial organisms evolve yet, we used, as benchmarks, genomes shaped by evolution during experiments executed with the CPU implementation of the model. These organisms evolved with 3 different mutation rates (10^{-4}, 10^{-5}, 10^{-6}), giving them different genome structures (size, density of coding part, ...) [14], with a population size of 1,024 individuals. At regular periods of time, we pick one genome of the population for each evolution to have different genomes from generations 100 to one million. Then for each collected genome, we measure 20 times the evaluation process duration of a population composed of clones (*i.e.*, initialized with the same genome). The population size goes from 1,024 to 32,768 (2^{10} to 2^{15}) individuals. These measures are done on both GPUs and for each implementation mentioned. To compare with the CPU, we also ran these measures with the same inputs but on the CPU implementation of the evaluation process. This implementation is a bit different because of the adaptations made for the GPU but these measures can tell us an overview of what is gain with the GPU port. The CPU used is an AMD EPYC 7262 running at 3.2 GHz.

4.1 Detailed implementation

The section 3.2 presents two major problems encountered during the development and the different algorithmic solution to solve them. The following subsections will compare the different algorithms for the related kernels: *Search Patterns* and *Find Genes*. These kernels and the related algorithms propose solutions to expose intra-individual parallelism. Consequently, results are only presented for one population size of 4,096. Indeed, inter-individual parallelism

[2] Indeed, for RNAs, we look for the terminator that is the closest from the promoters. For genes, we look for all the gene-starts that are within the RNA range.

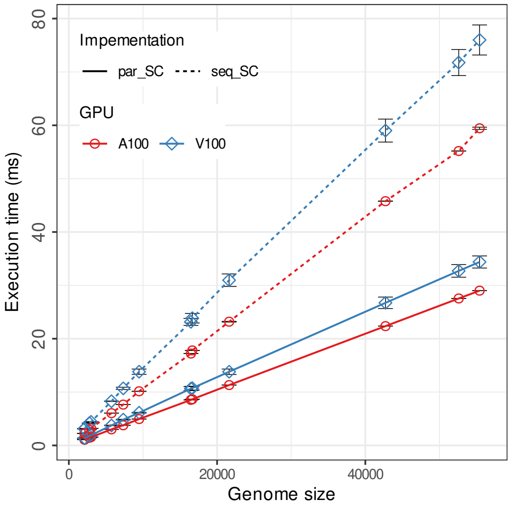

Figure 2: Execution time for the different implementations of *Search Patterns* kernel depending on the genome size of individuals. Population size is 4,096 cloned individuals. The results compare the A100 and V100 GPU.

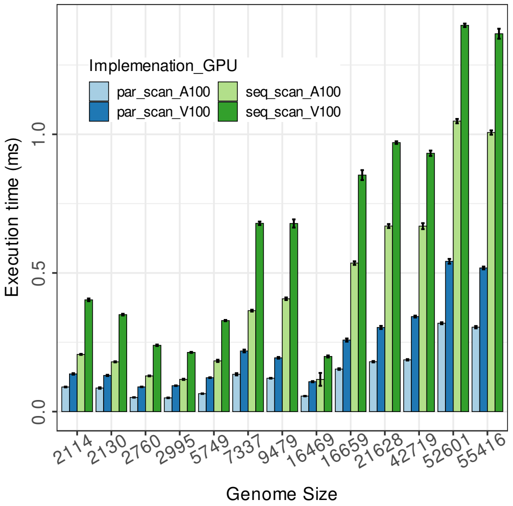

Figure 3: Execution time of the different implementations of *Find Genes* kernel depending on the genome size of individuals. Population size is 4,096 cloned individuals. The results compare the A100 and V100 GPU.

remains the same (*i.e.*, one block per individual regardless of the size of the population).

4.1.1 Stream compaction of metadata in Search Patterns. Figure 2 shows the comparison of the two algorithms for the Stream Compaction problem: `seq_SC` with the sequential compaction using a boolean arrays and `par_SC` with a compaction running on the fly during pattern searching thanks to atomic operations. On both implementation, the pattern searching is done in parallel. The figure clearly shows that atomic operations of the GPU are a major advantage in our case, allowing the `par_SC` implementation to be in average 2.1 times faster on both GPUs. This improvement is really important for Aevol because it is the most time consuming step of the evaluation. Comparing `par_SC` between the two GPUs, the most recent A100 is in average 19% faster than the V100.

4.1.2 Scanning genes in Find Genes. Figure 3 shows the comparison of the two implementations of the algorithm allowing to extract nested genes data to a non nested array on both GPUs. `seq_scan` algorithm is sequentially browsing all the RNAs. `par_scan` algorithm is applying a two step approach: first, a scan operation and then, the processing of RNAs in parallel. Unlike *Search Patterns* which algorithm complexity is tightly linked with the genome size, for *Find Genes* the complexity is linked to the amount of RNAs and genes inside the genome. Evolution makes the density of RNAs change depending on the generation and the mutation rate. This is why Figure 3 does not present the result as curves. Anyway, the results show that `par_scan` is in average 2.8 times faster on both GPU but loose efficiency when the density of RNAs drops. It is interesting to notice that the scan algorithm adds computational work to the program but allows for more parallelism. This is an important principle to keep in mind with parallelism. Adding work for the sake of parallelism can be profitable if the critical path of the algorithm is reduced. Comparing `par_scan` between the two GPUs, the most recent A100 is in average 41% faster than the V100.

4.2 CPU against GPU

We can now check the overall performance of our GPU port. Figure 4 shows the speed up of the NVIDIA A100 against the sequential evaluation process of Aevol. It shows that the GPU implementation can go up to 1,000 times faster to evaluate a population. The speed up depends a lot of the size of the population but is no lower than 370. This is a huge improvement and shows the impact of this two-level parallelization. The lower performance with smaller population could indicate that we do not saturate the computational capabilities of the GPU. Execution of some warps does not manage to overlap the data loading of other warps, putting multiple warps in a waiting state. With larger population, there are more blocks allocated for each SM and then more warps per SM. The *Search Patterns* kernel must be the most subject to it as it is the most memory bound kernel. Figure 5 shows the difference of performance between the NVIDIA V100 and A100 GPUs with a clonal population. As expected, the A100 is faster than the V100 with 35% more SMs (from 80 to 108), however the speedup is increased by more than 50% for large populations. It means that the upgrade of micro-architecture improved the power of the GPU on other aspect than just parallel computation. Indeed, cache memory and memory bandwidth are larger on the Ampere architecture.

4.3 Heterogeneous Population

For all the previous results, we execute the evaluation on a clonal population. However, such a population does not represent a typical Aevol experiment. Indeed, the evolution process implies stochastic mutations that will change the genome size of some individuals. Some are going to be larger, other smaller. Actually, you can have

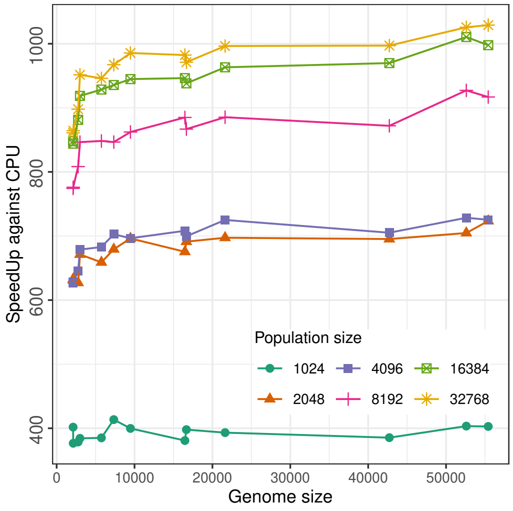

Figure 4: Speedup against sequential CPU depending on the genome size. Measures are from the NVIDIA A100 GPU. Shapes and colors depends on the clonal population size.

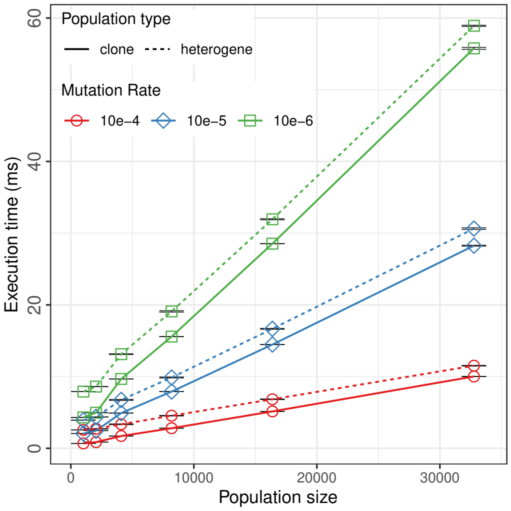

Figure 6: Execution time depending on the size of different types of population. Shapes and colors are for different mutation rate, and line types are for either clonal of heterogeneous population. Execution time variations (out of the 20 repetitions) are very low as shown by error bars. Measures are from the NVIDIA V100 GPU.

several orders of magnitude between the largest and the smallest individual during one generation. Most of the population will have a similar genome size, but it have been shown [26] that these differences bring scheduling issues on CPU. Because our implementation allocates one block per individual, no matter its size, such scheduling issue should happen.

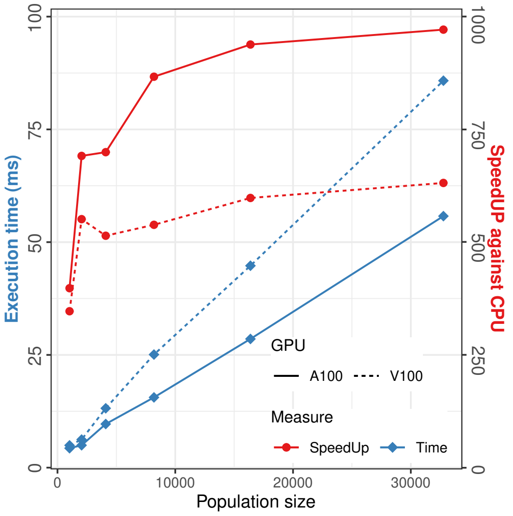

Figure 5: Execution time and SpeedUp against sequential CPU on two different GPUs (A100 and V100) depending on the population size. Individuals are clones with genome size of 16659.

Figure 6 shows the difference of performance between a clonal population and a more realistic heterogeneous population. The heterogeneous ones come from the initial CPU experiments with the population sets to 1,024 individuals. To increase the population size, we just copy the population several times. We compare these heterogeneous populations against one organism that have a genome size close to the mean genome size of the population. Figure 6 shows that on heterogeneous population, performance are lower and this is especially true with small population. The scheduling problem is solved on the GPU by a List Scheduling algorithm [9] that perform better with an increasing size of the list to schedule. To improve performance, we could try to allocate blocks according to the size individuals' genome making the blocks equal in computational need and removing the scheduling problem. This will, however, bring other algorithmic and synchronization problems.

5 RELATED WORK

As genetic algorithms involve population with a genome, parallelizing such a program can be self-evident. In [5], a general method to parallelize genetic algorithms on GPU is given and a classification on these methods is proposed. The authors clearly state that parallelization of genetic algorithms is of interest, but doing it for GPUs is not obvious at all. They proposed two types of parallelism level. One involving one thread per individual and one involving one block (or more generally one group of threads) per individual. This second kind is similar to the two level parallelism we have used.

Using this classification, the authors of an early GPU parallelization of AutoDock [12] with CUDA compared this two kind of parallelism. With small population, the per block implementation outperforms by far the per thread because GPU cores were more saturated with instructions reaching nearly a speedup of 50 against sequential CPU with an NVIDIA Tesla C1060. Unfortunately, one part of their per block algorithm did not fully used parallelism, and the per thread implementation was better for larger population size.

With a more recent GPU implementation of AutoDock4 [25] using OpenCL, they use this per block implementation benefiting of the second level of parallelism to evaluate individuals. They reached a speedup of about 157 against sequential CPU with an NVIDIA GTX 1080 Ti.

With this other recent genetic algorithm to solve the capacitated vehicle routing problem [1], the authors used a mix of the two parallelism types, sometimes using par thread parallelism and sometimes per block. They reached a speedup of 454 with a NVIDIA GeForce RTX 2080. It is interesting to see that speedup increases while you increase the genome size.

Applying this classification to X-Aevol, our strategy is a per block parallelism using the available threads within the block to process the individual genome in parallel achieving speed up of 1,000 with an NVIDIA A100 for the evaluation process. This speed up is against a CPU implementation of Aevol.

As for other *in-silico* experimentation evolution software, their genome model differs a lot from Aevol and does not target accurate representation of real-life genomes. In Avida [22], genomes are programs composed of a set of simple instructions that can self replicate if executed. In EcoSim[10], the genome is a matrix representing a behavioral model. All individuals have a unique behaviour and for a single generation, they play a game of prey/predator. However, unlike Aevol, these model does not have any GPU implementations. At the best of our knowledge, X-Aevol is the only *in-silico* experimentation evolution software that has a GPU implementation.

6 CONCLUSION AND FUTURE WORK

In this paper, we show that, by using the power of a GPU, we managed to massively accelerate the evaluation process of Aevol, a genetic algorithm designed to make *in-silico* evolutionary experiments. With a speed up of up to 1,000 against a sequential CPU implementation, it proved the potential of GPUs to go beyond our current computing capabilities. However this paper also showed that this is not an easy task and that algorithms have to be re-designed to match this massive parallelism.

Our work is then a successful GPU port of a program conveying irregular structures of data with variable size thanks to different parallel algorithms and their implementation using advanced hardware operations.

Our experimental setup relies on populations built to control the heterogeneity of the genomes. The main interest is to let possible to generate worst and best scenarios to measure performance of X-Aevol. Future work is to make real simulation in order to simulate the full evolution of an artificial organism.

Another point of interest is the ability to execute our GPU port on other vendor GPUs than the ones from NVIDIA. As CUDA is a proprietary parallel computing platform, it cannot be used for AMD's or Intel's GPUs. Frameworks and languages[3][4][5] have emerged recently to unify the development of parallel computing to use different kinds of accelerators with the same base code while maintaining a high performance portability.

Last but not least, studies show the impact of the size of populations on the genome size and structures [4, 20]. Accordingly, Aevol can be required to simulate very large population exceeding million individuals. To do so, the computing power of a single GPU will not be enough. We would have to work on multi-GPUs implementation using partitioning algorithms that will take into account the micro architectural properties of GPU and our inner knowledge of the biological model of Aevol to cut the overall population into smaller ones assigned to different GPUs.

[3] https://github.com/ROCm-Developer-Tools/HIP

[4] https://github.com/kokkos

[5] https://www.oneapi.com/

REFERENCES

[1] Marwan F. Abdelatti and Manbir S. Sodhi. 2020. An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 663–671. https://doi.org/10.1145/3377930.3390159

[2] Mark"us Billeter, Ola Olsson, and Ulf Assarsson. 2009. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana) *(HPG '09)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/1572769.1572795

[3] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Technical Report. Synthesis of Parallel Algorithms.

[4] Brian Charlesworth and Nick Barton. 2004. Genome Size: Does Bigger Mean Worse? *Current Biology* 14, 6 (2004), R233–R235. https://doi.org/10.1016/j.cub.2004.02.054

[5] John Runwei Cheng and Mitsuo Gen. 2019. Accelerating genetic algorithms with GPU computing: A selective overview. *Computers & Industrial Engineering* 128 (Feb. 2019), 514–525. https://doi.org/10.1016/j.cie.2018.12.067

[6] Yihui Cui, Yan Yang, Zheyi Ni, Yiyan Dong, Guohong Cai, Alexandre Foncelle, Shuangshuang Ma, Kangning Sang, Siyang Tang, Yuezhou Li, Ying Shen, Hugues Berry, Shengxi Wu, and Hailan Hu. 2018. Astroglial-Kir4.1 in Lateral Habenula Drives Neuronal Bursts to Mediate Depression. *Nature* 554 (Feb. 2018), 323–327. https://doi.org/10.1038/nature25752

[7] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU memory allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 27–37. https://doi.org/10.1145/3293883.3295727

[8] Ajai V. George, Sankar Manoj, Sanket R. Gupte, Sayantan Mitra, and Santonu Sarkar. 2017. Thrust++: Extending Thrust Framework for Better Abstraction and Performance. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 368–377. https://doi.org/10.1109/HiPC.2017.00049

[9] R. L. Graham. 1966. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal* 45, 9 (1966), 1563–1581. https://doi.org/10.1002/j.1538-7305.1966.tb01709.x

[10] Robin Gras, Didier Devaurs, Adrianna Wozniak, and Adam Aspinall. 2009. An Individual-Based Evolving Predator-Prey Ecosystem Simulation Using a Fuzzy Cognitive Map as the Behavior Model. *Artificial Life* 15, 4 (Oct. 2009), 423–463. https://doi.org/10.1162/artl.2009.Gras.012

[11] Benjamin C. Haller and Philipp W. Messer. 2019. Evolutionary Modeling in SLiM 3 for Beginners. *Molecular Biology and Evolution* 36, 5 (May 2019), 1101–1109. https://doi.org/10.1093/molbev/msy237 Publisher: Oxford Academic.

[12] Sarnath Kannan and Raghavendra Ganji. 2010. Porting Autodock to CUDA. In *IEEE Congress on Evolutionary Computation*. 1–8. https://doi.org/10.1109/CEC.2010.5586277 ISSN: 1941-0026.

[13] C. Knibbe, G. Beslon, V. Lefort, F. Chaudier, and J. M. Fayard. 2005. Self-adaptation of Genome Size in Artificial Organisms. In *Advances in Artificial Life*, Mathieu S. Capcarrère, Alex A. Freitas, Peter J. Bentley, Colin G. Johnson, and Jon Timmis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 423–432.

[14] Carole Knibbe, Antoine Coulon, Olivier Mazet, Jean-Michel Fayard, and Guillaume Beslon. 2007. A Long-Term Evolutionary Pressure on the Amount of Noncoding DNA. *Molecular Biology and Evolution* 24, 10 (08 2007), 2344–2353. https://doi.org/10.1093/molbev/msm165 arXiv:https://academic.oup.com/mbe/article-pdf/24/10/2344/13639022/msm165.pdf

[15] Carole Knibbe and David P. Parsons. 2014. What happened to my genes? Insights on gene family dynamics from digital genetics experiments. In *ALIFE 14 (14th Intl. Conf. on the Synthesis and Simulation of Living Systems)*, H. et al. Sayama (Ed.). MIT Press, New York, NY, United States, 33–40. https://doi.org/10.7551/978-0-262-32621-6-ch006

[16] Scott LeGrand, Aaron Scheinberg, Andreas F. Tillack, Mathialakan Thavappiragasam, Josh V. Vermaas, Rupesh Agarwal, Jeff Larkin, Duncan Poole, Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas Koch, Stefano Forli, Oscar Hernandez, Jeremy C. Smith, and Ada Sedova. 2020. GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer: Porting, Optimization, and Application to COVID-19 Research. In *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics* (Virtual Event, USA) *(BCB '20)*. Association for Computing Machinery, New York, NY, USA, Article 43, 10 pages. https://doi.org/10.1145/3388440.3412472

[17] Richard E. Lenski. 2017. Experimental evolution and the dynamics of adaptation and genome evolution in microbial populations. *The ISME Journal* 11, 10 (Oct. 2017), 2181–2194. https://doi.org/10.1038/ismej.2017.69 Number: 10 Publisher: Nature Publishing Group.

[18] Richard E Lenski, Jeffrey E Barrick, and Charles Ofria. 2006. Balancing Robustness and Evolvability. *PLOS Biology* 4, 12 (12 2006), 1–3. https://doi.org/10.1371/journal.pbio.0040428

[19] Vincent Liard, David Parsons, Jonathan Rouzaud-Cornabas, and Guillaume Beslon. 2018. The Complexity Ratchet: Stronger than selection, weaker than robustness. *Artificial Life Conference Proceedings* 30 (July 2018), 250–257. https://doi.org/10.1162/isal_a_00051

[20] Michael Lynch and John S. Conery. 2003. The Origins of Genome Complexity. *Science* 302, 5649 (2003), 1401–1404. https://doi.org/10.1126/science.1089370

[21] Hubert Nguyen. 2007. *Gpu gems 3* (first ed.). Addison-Wesley Professional.

[22] Charles Ofria and Claus O. Wilke. 2004. Avida: A Software Platform for Research in Computational Evolutionary Biology. *Artificial Life* 10, 2 (April 2004), 191–229. https://doi.org/10.1162/106454604773563612

[23] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* 26, 1 (2007), 80–113. https://doi.org/10.1111/j.1467-8659.2007.01012.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x

[24] Sergio Peignier, Christophe Rigotti, and Guillaume Beslon. 2015. Subspace Clustering Using Evolvable Genome Structure. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 575–582. https://doi.org/10.1145/2739480.2754709

[25] Diogo Santos-Martins, Leonardo Solis-Vasquez, Andreas F Tillack, Michel F Sanner, Andreas Koch, and Stefano Forli. 2021. Accelerating AutoDock4 with GPUs and Gradient-Based Local Search. *Journal of Chemical Theory and Computation* 17, 2 (Feb. 2021), 1060–1073. https://doi.org/10.1021/acs.jctc.0c01006 Publisher: American Chemical Society.

[26] Laurent Turpin, Thierry Gautier, Jonathan Rouzaud-Cornabas, and Christian Perez. 2020. P-Aevol: An OpenMP Parallelization of a Biological Evolution Simulator, Through Decomposition in Multiple Loops. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems (Lecture Notes in Computer Science)*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 52–66. https://doi.org/10.1007/978-3-030-58144-2_4