A COMPARATIVE EVALUATION OF THE GPU VS. THE CPU FOR PARALLELIZATION OF EVOLUTIONARY ALGORITHMS THROUGH MULTIPLE INDEPENDENT RUNS

Anna Syberfeldt and Tom Ekblom

University of Skövde, Department of Engineering Science, Skövde, Sweden

ABSTRACT

Multiple independent runs of an evolutionary algorithm in parallel are often used to increase the efficiency of parameter tuning or to speed up optimizations involving inexpensive fitness functions. A GPU platform is commonly adopted in the research community to implement parallelization, and this platform has been shown to be superior to the traditional CPU platform in many previous studies. However, it is not clear how efficient the GPU is in comparison with the CPU for the parallelizing multiple independent runs, as the vast majority of the previous studies focus on parallelization approaches in which the parallel runs are dependent on each other (such as master-slave, coarse-grained or fine-grained approaches). This study therefore aims to investigate the performance of the GPU in comparison with the CPU in the context of multiple independent runs in order to provide insights into which platform is most efficient. This is done through a number of experiments that evaluate the efficiency of the GPU versus the CPU in various scenarios. An analysis of the results shows that the GPU is powerful, but that there are scenarios where the CPU outperforms the GPU. This means that a GPU is not the universally best option for parallelizing multiple independent runs and that the choice of computation platform therefore should be an informed decision. To facilitate this decision and improve the efficiency of optimizations involving multiple independent runs, the paper provides a number of recommendations for when and how to use the GPU.

KEYWORDS

Evolutionary algorithms, parallelization, multiple independent runs,GPU, CPU.

1. INTRODUCTION

Of the many approaches to parallelizing evolutionary algorithms, one of the most straightforward is multiple parallel independent runs of the same algorithm. Sudholt [1] showed that this approach significantly increased the success probability of an algorithm, that is, the probability of finding a satisfactory solution within a given time budget. The probability p of at least one successful run in γ in dependent runs is 1- 〖(1-p)〗 ^γ, where 〖(1-p)〗 ^γ is the probability of no run being successful. This phenomenon is commonly known as probability amplification. Besides increasing the success probability of an algorithm, the approach of multiple independent runs is also beneficial when setting up an experiment. Since no communication is needed during runtime, the configuration is very easy and the results need to be processed only after all runs have been completed.

The approach of independent runs for parallelizing an evolutionary algorithm can be used for different purposes. One example is when the optimization algorithm itself must be fast, which is the case when the computational time of the fitness evaluation is not dominant [2]. This is

DOI:10.5121/ijcsit.2017.9301

common when dealing with combinatorial optimization problems such as routing or assignment problems [2].

Combinatorial problems often involve a very fast fitness evaluation (such as, in the case of routing, summarizing the cost between a set of nodes), but require a very large number of algorithm iterations. Independent runs are also used for problems that benefit from a small population size. Can tú-Paz and Goldberg [3] have shown that for some optimization problems, the quality of the solutions found is higher with a small population run independently in parallel than with a large population given the same computational budget.

Independent runs are also used for parameter tuning. The behavior of an evolutionary algorithm is controlled by user-defined parameters (present in operators such as selection, crossover, and mutation).The settings of these parameters greatly affect the performance of the algorithm as they control the ratio between exploration and exploitation of the search space [4]. Finding the optimal settings for a particular problem is, however, problematic as it is seldom intuitive and requires considerable trial-and-error. Previous studies have shown that the algorithm must be run some ten thousand times with different parameters to perform parameter tuning [5].

To enable efficient, automatic parameter tuning, independent runs can be set up with different parameter values that are evaluated in parallel. The parameter space can thus be effectively explored and good parameter values can be found in a short time[6]. It has been suggested (e.g., [7] and [8]) that a graphics processing unit (GPU)can be used to efficiently run evolutionary algorithms in parallel. A GPU is a specialized processor originally designed to offload 3D graphics rendering from a microprocessor. Its architecture allows for massive parallelization through thousands of computing cores and the platform can be used to handle complex computations efficiently.

A video card with a GPU is now standard in most desktop computers and the low price of the GPU makes the platform available for anyone to utilize. Utilizing the GPU for parallelization of evolutionary algorithms has been discussed in numerous previous studies, and it has been shown that a GPU has clear advantages compared to a traditional CPU platform (for a comprehensive overview of these studies, see [9]). The reported speedup achieved with the GPU, however, varies greatly. Some studies claim that the speedup is over 7000 times [10], while other studies state that there is no way to reach speedups greater than an order of 100 [11].Some studies even state that the speedup is much lower than this, as low as 13 times [2].

There is obviously a lack of consensus in the research community on how efficient a GPU actually is in comparison with a CPU. This is especially true when it comes to parallel independent runs of evolutionary algorithms, for very few studies have investigated this topic. The vast majority of previous studies have focused on approaches in which the parallel runs are dependent on each other, as in the master-slave approach, the coarse-grained approach, and the fine-grained approach (e.g.[10] and[12]).The present study was prompted by the lack of common understanding of how efficient a GPU is in general for parallel independent runs of an evolutionary algorithm. The study was set up to investigate the performance of a GPU in comparison with a CPU and to analyze the gains of utilizing a GPU in the context of multiple independent runs.

Specifically, it aims to establish the conditions in which using a GPU is beneficial compared to using a CPU, and why. The study yields a number of concrete recommendations on when and how to use a GPU when implementing parallel independent runs of an evolutionary algorithm. The next section of this paper presents basic technical information about CPU and GPU. It is followed by an account of the experiments performed to evaluate and analyze the performance of a CPU versus a GPU. The results of these experiments are presented in section 4,and in section 5

these results form the basis for recommendations and guidelines for choosing between CPU and GPU. The conclusions drawn are summarized in the final section of the paper.

2. THE BASICS OF CPU AND GPU

A CPU is basically a microprocessor that executes instructions given by a program based on operations for arithmetic, control, logic, and input-output[13]. The fundamental components of a CPU include an arithmetic logic unit (ALU), processor registers that provide operations to the ALU and store the results, a control unit that executes instructions from memory, and various registers.A CPU is designed for low latency (quick response time) by implementing a large cache that is used for intermediate storage to avoid reading from global memory, which is very slow. Many computers today use a multi-core processor, which is a single chip containing two or more CPUs called “cores”. With multiple cores, it is possible to run multiple processes in parallel and thus speed up the execution time considerably.

The purpose, and thus also the design, of a CPU differs greatly from that of a GPU. Originally, the GPU was made for rendering images in computer games and was optimized for the fact that each pixel value can be processed independently from the others[14]. The development of the GPU is still to a large extent driven by the gaming industry, but today the GPU is also extensively used in a broad range of other application domains and for general purpose computations (see[15]).Instead of focusing on low latency as a CPU does, a GPU focuses on high throughputs that is, calculating on as much data as possible. Thus a GPU has more hardware allocated to computation and less to fast cache memory, the opposite of a CPU. Because of these differences, a CPU is good at processing serial instructions that use a lot of memory, while a GPU is good at processing instructions in parallel that use little memory. Another main difference between a CPU and a GPU is that a CPU has a few powerful cores while a GPU has thousands of weaker cores. This gives the GPU an advantage in highly parallelizable applications, while the CPU is better for sequential runs. An overview of the structural differences between CPU and GPU is provided in Figure 1 below.

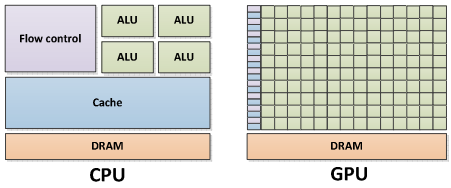


Figure 1. CPU architecture (to the left) and GPU architecture (to the right), adapted from [16]

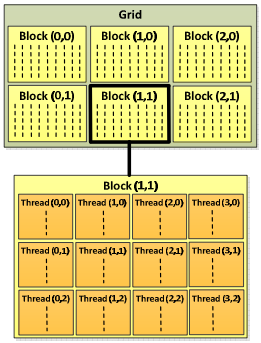


Figure 2. Illustration of blocks and threads on a GPU, adopted from [16].

Code written for a CPU cannot be run on a GPU since the two platforms are based on different hardware and thus require code specifically written for each specific platform. For NVIDIA graphics cards the language is CUDA, which is accessible as a superset of C++, among others. From a programmer's perspective, CUDA code looks like normal C++ code, except that functions are marked as being run able on CPU, GPU, or both. The code run on the GPU is called a "kernel." The difference from code run on the CPU is that the each kernel will be executed by thousands, or even millions, of threads. The programmer can specify the number of blocks and the number of threads per block to be run. The threads in a block will all run on the same core. An overview of the concepts is given in Figure 2 below. The next section of this paper describe show the CPU and the GPU were evaluated in this study, based on a number of practical experiments.

3. EXPERIMENTS

A number of experiments were performed to evaluate and analyze the performance of the CPU and the GPU. Subsection 3.1 describes the setup of these experiments, starting with a presentation of the selected evolutionary algorithm, while Subsection 3.2 describes the implementation of the algorithm. The optimization problems used in the evaluation are presented in Subsection 3.3, and the chapter ends with a description of the experimental platform in Subsection 3.4.

3.1 Evolutionary algorithm

There are numerous evolutionary algorithms. For this study, the well-known elitist non-dominated sorting genetic algorithm (NSGA-II) was chosen as it has been used to solve complex problems in various domains over the years. It should, however, be noted that the choice of algorithm is of minor importance for this study as it is not the algorithm itself that is being investigated. NSGA-II is basically a genetic algorithm with features for handling multiple trade-off solutions[17], making it well suited for use with real-world problems involving the simultaneous optimization of multiple objectives. In order to handle multiple, conflicting optimization objectives, the algorithm uses a Pareto approach to search for the set of best trade-off solutions. Pseudo code for NSGA-II is shown below.

```
generationsCount = 0
currentPopulation = GenerateRandomIndividuals(N)
EvaluateFitness(currentPopulation)
NonDominatedSort(currentPopulation)
while(generationsCount < X)
    while(newPopulation.count < N)
        parent1, parent2 = Select(currentPopulation)
        child1, child2 = Crossover(parent1, parent2)
        Mutate(child1, child2)
        newPopulation.Add(child1, child2)
    nextGeneration = currentPopulation + newGeneration
    NonDominatedSort(nextGeneration)
    currentGeneration = SelectBestN(newGeneration)
    generationsCount = generationsCount + 1
```

For a detailed description of the algorithm's implementation, see [18]. The NSGA-II in this study is implemented with a binary tournament for the selection of parents. This selection operator works by picking two individuals randomly from the current generation and then selecting the better one based on rank (or crowding distance, if they have the same rank). The procedure is repeated twice in order to produce two parents. For the crossover, a one-point split crossover operator is implemented. This operator works by picking one point randomly. The first

child is created by using all the genes to the left of this point from the first parent, and to the right of the second parent. The opposite process is then used to create the second child, which means that genes are taken to the right of the point from the first parent and to the left of the second parent. For the mutation, a Gaussian mutation method is chosen. This method works by offsetting a random gene by a random number taken from a normal distribution. Table 1 below presents the default settings for all parameters implemented in the algorithm. All values were carefully selected based on a thorough analysis.

Table 1. Default parameter settings

Parameter	Setting
Number of generations	100
Genome size	20
Population size	50
Crossover probability	1.0
Mutation size	Clamped between 0 and 1, mean = 0 and standard deviation =0.33
Mutation probability	1 divided by the genome size

3.2. Algorithm implementation for CPU and GPU

C#

```
while (newGeneration.Count < populationSize)
{
       IIndividual<T> parent1 = selector.Select(currentGeneration);
       IIndividual<T> parent2 = selector.Select(currentGeneration);

       IIndividual<T> child1;
       IIndividual<T> child2;
       crossover.Crossover(parent1, parent2, out child1, out child2);

       mutator.Mutate(child1);
       mutator.Mutate(child2);

       child1.Objectives = evaluator.CalculateObjectives(child1);
       child2.Objectives = evaluator.CalculateObjectives(child2);

       newGeneration.Add(child1);
       newGeneration.Add(child2);
}

newGeneration.AddRange(currentGeneration);

newGeneration = sorter.GetBestIndividuals(newGeneration, populationSize);
```

CUDA C++

```
i = 0;
while(i <popCount)
{
       int p1 = TurnamentSelection(pop);
       int p2 = TurnamentSelection(pop);

OnePointSplitCrossover(pop[p1], pop[p2], c1, c2);

       GaussianMutation(c1, mutationChance);
       GaussianMutation(c2, mutationChance);

       ZDT1(c1);
       ZDT1(c2);

       newPop[i++] = c1;
       newPop[i++] = c2;
}

j = 0;
while(j <popCount)
       newPop[i++] = pop[j++];

GetBestIndividuals(newPop, pop);
```

Figure 3. NSGA-II implementation in C# (to the left) and CUDA C++ (to the right)

The CPU implementation is written in C# while the GPU implementation is written in CUDA C++. The algorithms were first implemented in C# and then in CUDA C++ so that the higher level code of C# could act as a reference guide to the lower level of CUDA C++. The implementation used version 4.5 of the .NET Framework and used CUDA version 2.0. The implementation of NSGA-II in C# and in CUDA C++, respectively, is shown in Figure 3 below. The implementations are as similar as possible and only deviate where absolutely necessary due to language or hardware aspects (C# uses interfaces and generics, which are not used in CUDA C++). It can be noted that the non-domination sort of reduced computational complexity suggested by Jensen [19] is implemented in both versions of the algorithm in order to speed up the algorithm.

The implementation of the one-point split crossover is shown in Figure 4 below.

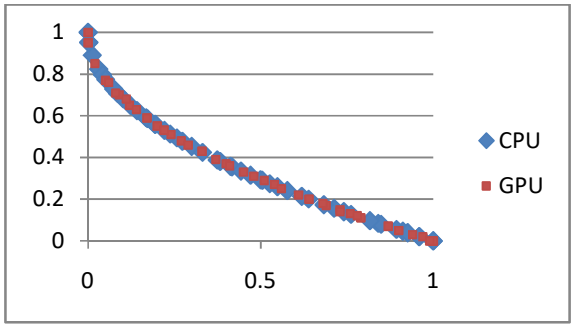
C#	**CUDA C++**

`int split = rnd.Next(p1.Genome.Length);`
 	
`c1 = p1.DeepCopy();`
`c2 = p2.DeepCopy();`
 	
`for (int i = 0; i < split; i++)`
`{`
`  c2.Genome[i] = p1.Genome[i];`
`  c1.Genome[i] = p2.Genome[i];`
`}`
	`int split(curnd(genomeCount));`
	
	`c1 = p1;`
	`c2 = p2;`
	
	`for (int i = 0; i < split; i++)`
	`{`
	`  c2.Genome[i] = p1.Genome[i];`
	`  c1.Genome[i] = p2.Genome[i];`
	`}`

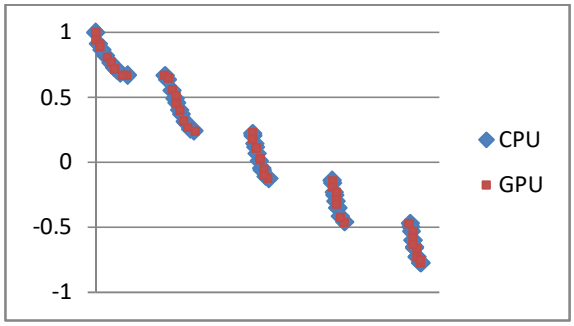
Figure 4.Implementations of one-point split crossover

3.3 Optimization problems used in the evaluation

In the field of evolutionary optimization, it is common to use standardized benchmark problems to assess the relative performance of different algorithms. These problems enable the comparison and replication of experiments, and are also considerably faster to run than real-world problems. A set of guidelines for the systematic development of benchmark problems for multi-objective optimization was first proposed in [20]. Based on these guidelines, Zitzler et al. [21] suggested a number of benchmark functions, known as the "ZDT problems," that have been extensively used in the literature for the analysis and comparison of multi-objective evolutionary algorithms.



ZDT1 (upper)

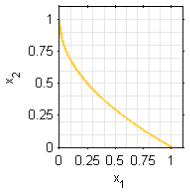
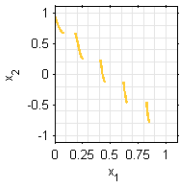


ZDT3 (lower)

Figure 5.Results from verification of the implementation of ZDT1 (upper) and ZDT3 (lower).

These problems have properties that are known to cause difficulties in converging to the true Pareto-optimal front and reflect characteristics of real-world problems such as multimodality, non-reparability, and high dimensionality. In this study, ZDT1 and ZDT3 are used to test the optimization algorithm. The details of these problems are presented in Table 2 below. In the study, both problems were implemented with 20 input parameters.

Table 2. ZDT1 and ZDT3 benchmark problems used in the evaluation.

Function	Variable bounds	Objective functions	Optimal solutions	Optimal Pareto front
ZDT1	[0,1]	$f_1(x) = x_1$ $f_2 = g * \left(1 - \sqrt{\frac{x_1}{g}}\right)$ $g(x_2, \ldots x_n) = 1.0 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$	$x_1 \in [0,1]$ $x_i = 0$ $i = 2, \ldots, n$	
ZDT3	[0,1]	$f_1(x) = x_1$ $f_2 = g * \left(1 - \sqrt{\frac{x_1}{g}} - \frac{x_1}{g}\sin 10\pi x_1\right)$ $g(x_2, \ldots x_n) = 1.0 + \frac{9}{n-1}\sum_{i=2}^{n} x_i$	$x_1 \in [0,1]$ $x_i = 0$ $i = 2, \ldots, n$	

To verify that the implementation of NSGA-II in C# and CUDA C++, respectively, was correct, the algorithms were run on the ZDT problems for 100 generations with a population size of 50

and genome size of 20. Results from the verification are shown in Figure 5 below and confirm that the implementations were correct.

3.4 Experimental platform

The experiments were run on a Windows 7 Professional x64 computer with a 4GHz Intel i7-4790K CPU, 16GB RAM, and an NVIDIA Ge Force 980 graphics card with 4GB VRAM. This graphic card represented the latest GPU architecture at the time of the study. The C# and CUDA C++ implementations were run with exactly the same settings with respect to algorithm parameter values (see Table 1).

4. EVALUATION OF THE CPU VS. THE GPU

The performance of the CPU versus the GPU was evaluated from two perspectives: (1) how the computational cost(time) grows with the number of parallel instances (number of instances of NSGA-II that are run concurrently), and (2) how the computational cost grows with the amount of data handled. The amount of data was changed by varying the genome size, for the size of the genome greatly affects the amount of data handled by the algorithm but barely affects the computation cost. The opposite applies to the number of parallel instances; the number of parallel instances does not affect the amount of data handled but greatly affects the computational cost.

As described in section 2, the main difference between the CPU and the GPU in terms of performance is that the former is good at processing serial instructions that use a lot of memory, while the latter is good at processing instructions in parallel that use little memory. Thus evaluating performance based on the number of parallel instances as well as the amount of data handled ensures a fair and relevant comparison between the CPU and GPU.

Subsection 4.1 deals with the evaluation with respect to parallel instances, while subsection 4.2 deals with the evaluation based on the amount of data handled. Since the evaluation results from the ZDT1 and ZDT3 test problems are virtually identical for all experiments, graphs for only one of the problems (ZDT3,chosen at random) are shown in the presentation of the results.

4.1Computational cost in relation to the number of parallel instances

To evaluate how the computational cost grows with the number of parallel instances, the CPU and GPU implementations were run with a vast number of parallel instances, ranging from one instance up to 50,000 instances.

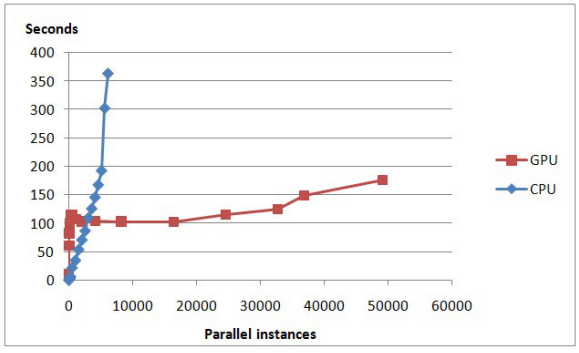


Figure 6.Computational cost in seconds as a function of the number of parallel instances for CPU and GPU.

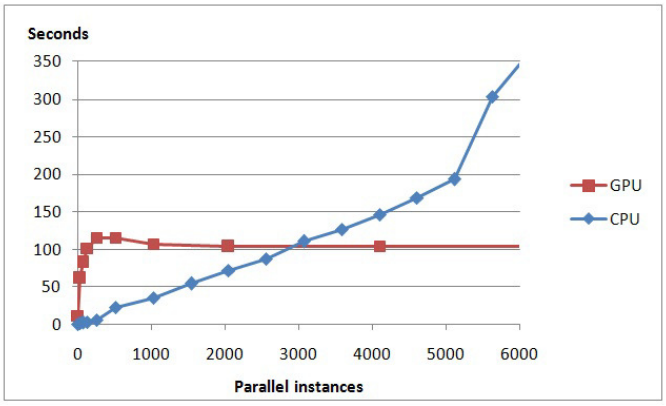


Figure 7. Zoomed-in view of graph presented in Figure 6.

4.2 Computational cost in relation to the amount of data handled

To evaluate how the computational cost grows with the amount of data handled, the CPU and GPU implementations were tested withgenome sizes rangingfrom 2 to 30.Varying the genome size is very easy with ZDT test problems as the number of input parameters can be set arbitrarily.Each genome size was run with256 parallel instances.The results of the experiments are shown in Figure 8 below. It is clear that the CPU is considerably more efficient than the GPU when it comes to handling data. The computational cost with the CPU stays the same regardless of the amount of data handled, while the computational cost of the GPU grows as the amount of data grows.

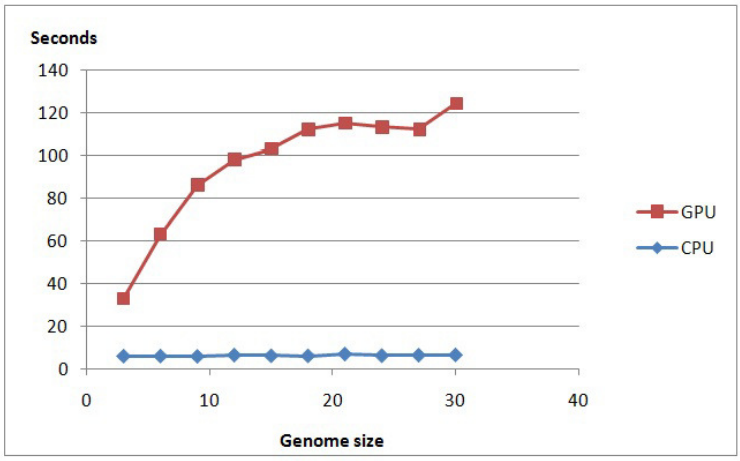


Figure8.Computational cost in seconds as a function of the amount of data handled (genome size).

In the light of these results, it is interesting to investigate whether the amount of data handled (genome size) has any effect on the breakpoint that was identified in Figure 7 at which the GPU became better than the CPU. To evaluate this, exactly the same experiment performed in section 4.1 was run once again but this time with a much smaller amount of data – the genome size was set to 2 instead of 20.The result is shown in Figure 9 below (zoomed in to the interesting part of the graph).

In this case, too, the CPU starts out better, but the breakpoint now comes earlier, at 1700 instances rather than 3000. This indicates that the breakpoint comes earlier when a smaller amount of data is being handled.

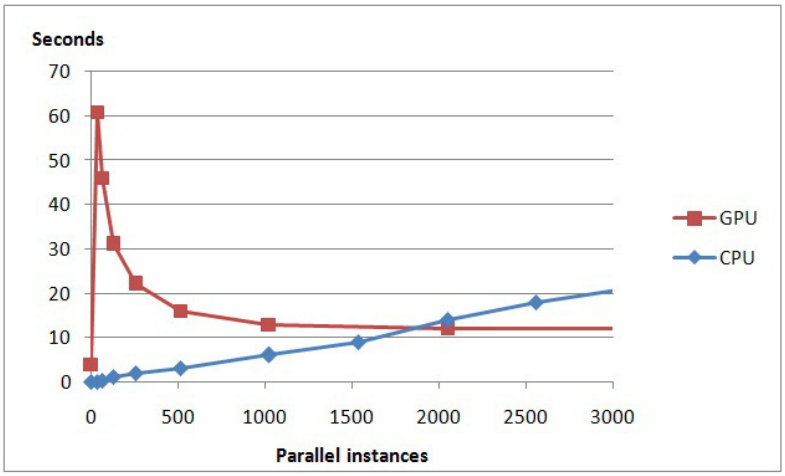


Figure9.Computational cost in relation to the number of parallel instances when a small amount of data is handled.

The hump in the computational cost of the GPU at very small numbers of parallel instances seen in Figure 7 is due to the fact that the GPU does not like to be idle. Below 1024 parallel instances, the GPU cannot fill one thread block,which results in idling and is associated with driver and memory management overheads that are computationally costly. The overhead arises because the program needs to use driver code to communicate with the GPU and spends time copying data back and forth from the GPU.

5. RECOMMENDATIONS ON HOW TO CHOOSE BETWEEN CPU AND GPU

The results of the experiments highlight a number of important aspects to consider when choosing whether to use CPU or GPU. The following five general recommendations are intended to assist users in making the right decision about parallelization platform and so lead to more efficient optimizations.

Make an informed decision whether to use the GPU or CPU

Many previous studies have suggested that a GPU is superior to a CPU, and it is easy to believe that the GPU should be used by default. However, as this study shows, a GPU is not always superior to a CPU; the opposite is sometimes the case, It is thus important to make a conscious choice about which platform to use. This might seem like an obvious recommendation, but scanning published articles in which a GPU was used for parallelization of evolutionary algorithms indicates that the decision to use a GPU was often not informed at all, but more a case of following the herd.

Analyze the amount of data handled in the specific optimization problem

As shown in the experiments, the breakpoint at which the GPU outperforms the CPU depends on the amount of data handled. With a normal genome size, the CPU was most efficient up to 3000 parallel instances in the experiments, but with a really small genome size, the breakpoint came at

a mere 1700 parallel instances. It is clear that the amount of data handled has a large effect on the efficiency of the GPU. It is therefore important to analyze the amount of data being handled in the optimization problem at hand and let this information, in combination with the number of parallel instances available, guide the decision whether to use the GPU or CPU.

Make sure you use the latest GPU graphic card

To achieve maximum benefits from the GPU, it is of uttermost importance to select a graphic card with the latest GPU architecture. To show the importance of this, we performed the same experiment described in section 4.1 using an older GPU graphic card, namely an NVIDIA NVS 310 with 512MB VRAM. Only the graphic card was changed; all other settings were exactly the same as in section 4.1. The results of this experiment are presented in Figure 10 below. For comparison, we also include the previously presented GPU and CPU results in the graph (note that the previous GPU results were generated with the latest GPU graphic card). As Figure 10 shows, there is a tremendous difference between the performance of an old and a new GPU graphic card. It is actually much better to use a CPU than an older GPU graphic card, for the older card could not handle more than approximately 2500 parallel instances – the operating system shut down the card driver due to overload at around 2500 parallel instances.

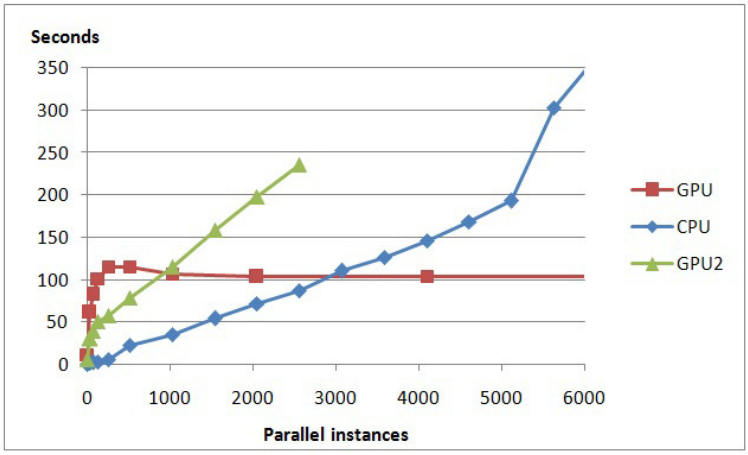


Figure 10.Comparison of the performance of an older GPU
Graphic card (called GPU2 in diagram).

The following web site offers tips on how to select the best GPU graphic card: http://www.pcgamer.com/how-to-buy-a-graphics-card-six-things-you-must-know-about-gpus/.

Program your GPU as a GPU

Writing program code for a GPU is completely different from writing traditional code for a CPU. If code for the GPU is written in the same way as for a CPU, the CPU will always outperform the GPU. It is therefore of utmost importance that you program your GPU as a GPU and not as a CPU. Most researchers within the optimization field are skillful programmers, but their competence in traditional programming is not enough to properly utilize the GPU. Therefore, make sure to learn how to program a GPU before starting to use it. Also make sure to investigate, and use, the abundance of tips and tricks available on the Internet about how to program a GPU efficiently. These will considerably enhance your expertise**.**

Tailor your problem and your implementation to the GPU

How you design your problem representation and your program code significantly affects the performance of the GPU. To achieve maximum performance, make sure to keep the characteristics of the GPU in mind when writing your program. The problem at hand needs to be as closely tailored to the GPU as possible. Some of the most important things to keep in mind are the following:

- GPUs achieve high performance by calculating many results in parallel. Thus, matrix and higher-dimensional array operations typically perform much better than operations on vectors or scalars. You can achieve better performance by vectorization, that is, by rewriting your loops to make use of higher-dimensional operations.
- By default, many operations in the program code are performed in double-precision floating-point arithmetic. However, today's GPUs and CPUs typically have much higher throughput when performing single-precision operations, and single-precision floating-point data occupies less memory. If your application's accuracy requirements allow the use of single-precision floating-point, it can greatly improve the performance of your code.
- Transferring data to and from a GPU is associated with a large overhead and can significantly hurt the performance of your application. In general, you should limit the number of times you transfer data to the GPU. If you can transfer data to the GPU once at the start of your application, perform as much of the calculation on the GPU as possible, and only transfer the results back into your application at the end, you will generally obtain the best performance.

6. CONCLUSIONS

Multiple independent runs of an evolutionary algorithm in parallel are often used to increase the efficiency of parameter tuning or to speed up optimizations involving inexpensive fitness functions, such as combinatorial problems involving routing or assignment. Researchers commonly adopt a GPU platform when implementing parallelization, for this platform has been shown to be superior to the traditional CPU platform in many previous studies. GPUs are very efficient, specialized processors originally designed for graphic rendering. Their highly parallel structure makes them effective for a range of complex computations. Since a video card with a GPU comes as standard in most desktop computers today, the platform is widely available at a low price.

Almost all existing studies on using GPUs for parallelization of evolutionary algorithms show that the GPU yields significant speedups compared to the CPU, but the reported speedup achieved varies considerably, ranging from an impressive speedup of over 7000 times [10] to a more modest speedup of 13 times [2].As a result it is unclear how efficient the GPU actually is in comparison with the CPU, especially for the parallelization of multiple independent runs, as very few studies have considered this topic. Instead, the vast majority of the previous studies comparing a CPU and a GPU focus on parallelization approaches in which the parallel runs are dependent on each other (such as master-slave, coarse-grained or fine-grained approaches).

The lack of studies comparing the performance of the GPU versus the CPU in the context of parallelizing an evolutionary algorithm using multiple independent runs motivated this study. A number of experiments were performed and different scenarios were analyzed in order to gain insights into the relative efficiencies of GPU and CPU. These insights were also compiled into number of recommendations for GPU use.

In summary, the results of the experiments showed that the amount of data and the number of available parallel instances have a decisive influence on which platform is more efficient. With larger amounts of data, the CPU is more efficient when a limited number of parallel instances are available (a few thousands), as using the GPU is associated with a significant overhead that negates the possible improvements gained by parallelization. With small amounts of data, the CPU is more efficient than the GPU, but only up to a relatively low number of parallel instances. Given a really large number of parallel instances, the GPU will, however, always win over the CPU regardless of the amount of data handled. However, even if there are a large number of parallel instances available, one should not blindly chose the GPU without also considering the programming effort involved. Programming a GPU requires learning a specialized programming language. Therefore, one should make sure that the performance gained by using the GPU justifies the amount of effort put into gaining the skills required to use the platform. It should, however, be noted that GPUs are constantly improving, and the CUDA language keeps getting better and better. As a result the entry level threshold for using the GPU will decrease over time.

One point that is not fully obvious, and therefore should be pointed out, is that the GPU platform is limited to optimization problems which can be represented mathematically. Many real-world optimization problems today are represented by simulations and approached by so-called simulation-based optimization [22]. Such problems cannot be parallelized using the GPU because the simulations are built in and executed through third-party software that cannot be run on the GPU (a GPU requires code specifically written for the platform).Researchers often want to parallelize problems approached by simulation-based optimization as they are computationally expensive. However, for these optimizations the CPU platform is the only choice and it is unnecessary to even consider using a GPU for this purpose.

Finally, the following recommendations emerge from this study:

- Make sure that your optimization problem can actually be implemented on a GPU.
- If you are going to parallelize your evolutionary algorithm through multiple independent runs, make an informed decision about which parallelization platform to use.
- Do not simply choose a GPU because it has proven superior in numerous previous studies –a traditional CPU sometimes outperforms a GPU.
- It is considerable simpler to develop software for the CPU platform.

As a concluding remark, it should be emphasized that the presented conclusions and recommendations are general and not specific for the GPU platform used in the study.

REFERENCES

[1] Sudholt D (2015) Parallel evolutionary algorithms. In: Springer Handbook of Computational Intelligence (eds. Janusz Kacprzyk and Witold Pedrycz), pp 929–959, Springer, Berlin Heidelberg. doi: 10.1007/978-3-662-43505-2_46

[2] Tsutsui S and Fujimoto N (2013) An analytical study of parallel GA with independent runs on GPUs. In Massively Parallel Evolutionary Computation on GPGPUs, Part of the Natural Computing Series,Springer-Verlag Berlin Heidelberg, pp 105–120

[3] Cantú-Paz E and Goldberg D (2003) Are multiple runs of genetic algorithms better than one? In: Proceedings of Genetic and Evolutionary Computation Conference, pp 801–812

[4] Črepinšek M, Liu S-H, Mernik M (2013) Exploration and exploitation in evolutionary algorithms: A survey. ACM Comput Surv 45(3):1–33, http://dx.doi.org/10.1145/2480741.2480752

[5] Smit S (2010) Parameter tuning of evolutionary algorithms. In: Applications of Evolutionary Computation. Springer, Berlin Heidelberg, pp 542–551

[6] Eiben A and Smit S (2011) Parameter tuning for configuring and analyzing evolutionary algorithms. Swarm and Evolutionary Computation 1(1):19–31, http://dx.doi.org/10.1016/j.swevo.2011.02.001

[7] Collet P (2013) Why GPGPUs for evolutionary computation? In: Massively Parallel Evolutionary Computation on GPGPUs, Natural Computing Series, pp 3–14. Springer, Berlin Heidelberg. ISBN: 978-3-642-37958-1

[8] Hofmann J, Limmer S, Fey D (2013) Performance investigations of genetic algorithms on graphics cards.Swarm and Evolutionary Computation 12:33–47. http://dx.doi.org/10.1016/j.swevo.2013.04.003

[9] Tsutsui S and Collet P (eds) (2013) Massively Parallel Evolutionary Computation on GPGPUs, Natural Computing Series, Springer, Berlin Heidelberg. ISBN: 978-3-642-37958-1

[10] Pospichal P, Jaros J, Schwarz J (2010) Parallel genetic algorithm on the CUDA architecture. In: Applications of Evolutionary Computation. Springer-Verlag Berlin, Heidelberg, pp 426–435

[11] Lee VW, Hammarlund P, Singhal R, Deisher M (2010) Debunking the 100X GPU vs. CPU myth. In: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA 2010, p 451. ACM Press, New York

[12] Wahib M, Munawar A, Munetomo M, Akama K (2011) Optimization of parallel genetic algorithms for nVidia GPUs. New Orleans, LA, IEEE Congress on Evolutionary Computation. doi:10.1109/CEC.2011.5949701

[13] Hennessy J and Patterson D (2011) Computer Architecture: A Quantitative Approach. Fifth Edition. Morgan Kaufmann Publishers, San Francisco. doi: 012383872X, 9780123838728

[14] Fernando R (2004) GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education.

[15] Fei X, Li K, Yang W, Li K (2016) CPU-GPU computing: Overview, optimization, and applications, In: Innovative Research and Applications in Next-Generation High Performance Computing, IGI Global. doi: 10.4018/978-1-5225-0287-6.ch007

[16] NVIDIA (2017) CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed 10January 2017.

[17] Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multi-objective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 2:182–197

[18] Deb K (2008) Multi-Objective Optimization using Evolutionary Algorithms. John Wiley & Sons, Chichester

[19] Jensen M (2003) Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. IEEE Transactions on Evolutionary Computation7(5)

[20] Deb K(1999) Multi-objective genetic algorithms: Problem difficulties and construction of test problems. Evolutionary Computation Journal 7(3):205–230

[21] Zitzler E, Deb K, Thiele L(2000) Comparison of multiobjective evolutionary algorithms: Empirical results. Evolutionary Computation Journal 8(2):125–148

[22] Fu M (2015) Handbook of Simulation Optimization. Springer, New York. ISBN 978-1-4939-1383-1