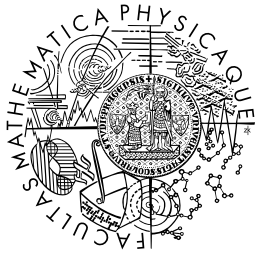


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Patrik Valkovič

GPU Parallelization of Evolutionary Algorithms

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Martin Pilát, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I wish to thank my supervisor, Mgr. Martin Pilát, Ph.D., for his guidance, help, and mostly patience during my work on this thesis. I am thankful for his friendly attitude and valuable advice.

I would also want to express my sincere gratitude to my family and friends, whose support was paramount for successful accomplishment of this thesis.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. I would like to thank CESNET for allowing me to use the computational power of MetaCentrum. I could not get results presented in this thesis without their support.

Title: GPU Parallelization of Evolutionary Algorithms

Author: Patrik Valkovič

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Graphical Processing Units stand for the success of Artificial Neural Networks over the past decade and their broader application in the industry. Another promising field of Artificial Intelligence is Evolutionary Algorithms. Their parallelization ability is well known and has been successfully applied in practice. However, these attempts focused on multi–core and multi–machine parallelization rather than on the GPU.
This work explores the possibilities of Evolutionary Algorithms parallelization on GPU. I propose implementation in PyTorch library, allowing to execute EA on both CPU and GPU. The proposed implementation provides the most common evolutionary operators for Genetic Algorithms, Real–Coded Evolutionary Algorithms, and Particle Swarm Optimization Algorithms. Finally, I show the performance is an order of magnitude faster on GPU for medium and big–sized problems and populations.

Keywords: Evolutionary Algorithms, Parallelization, GPU Computing, Genetic Algorithms, Particle Swarm Optimization

Contents

1. Introduction

Artificial Intelligence (AI) has become a phenomenon of these days. The latest advances in this field have achieved magnificent results and allowed the creation of systems and tools, which we would not think were possible thirty years ago. The biggest credit goes to the Artificial Neural Networks (ANN), which allowed this rapid and astonishing growth of the field over the past years. More precisely, the Deep Artificial Neural Networks (DANN) sparkled this process in 2011, when they started to exceed human performance on German traffic sign recognition benchmark [?].

Even though there had been known industrial applications of AI at the beginning of the 2000s, such as automatic check processing [?] or speaker recognition [?], it were mainly achievements in 2011 and 2012 which acquired attention from academia and the broader community. The progress in AI is moving forward day–by–day, and new results can be seen almost on a daily basis. The current state-of-the-art systems have already exceed human performance and traditional methods in a number of various tasks – like image recognition [?][?], object detection [?][?], video editing [?], question answering [?][?], natural language processing [?], and many more. Thanks to the rise of generative models, results in the field of super-resolution [?][?], image synthesis [?][?][?], text generation [?][?], and countless more amaze scientists by their accuracy and attention to details.

ANN also found their way into fields which traditional machine learning methods have not considered – fluid simulation [?][?], cloth simulation [?][?], or even whole physics [?][?] and movement of virtual entities [?][?]. Because of that, ANN are used in healthcare [?][?], Hollywood [?], robotics [?][?], and can even play computer games [?][?].

This long and incomplete list of successes would not be possible without the computation power of computers and data centers nowadays. In fact, the achievements of DANN in 2011 and 2012 were driven mainly by the effective implementation of Convolutional Neural Networks (CNN) in kernels and running them by the Graphical Processing Unit (GPU) [?]. Since then, the GPU and more recently the Tensor Processing Unit (TPU) have overcome traditional processors in the terms of speed and performance for ANN, and allowed these advancements in the field we can see now.

Another interesting field of Artificial Intelligence is evolutionary computation. Similarly to ANN, it performs parameters search in order to best solve the task at hand. Evolutionary algorithms and their variations have been successfully applied in a variety of fields – for example electronic circuits design [?][?], real-parameter optimization [?], combinatorial problems [?][?], robotics [?][?][?], design of neural networks [?][?], and many more.

There have been attempts to implement and run evolutionary algorithms on GPU [?] and use the computation power available. However, most of these methods focus on the Compute Unified Device Architecture (CUDA). CUDA and similar technologies have drawbacks – they use low-end languages like C or C++ and, therefore, require deep knowledge of the underlying hardware. The algorithms are also harder to modify.

The aim of this work is to analyze existing implementations of evolutionary

algorithms and propose their implementation on the GPU. The implementation needs to be easy to understand, extendable, and easily to modify. In order to meet the given criteria, implementation should reuse existing frameworks available for neural networks if possible.

The rest of this work is organized as follows. In the next Chapter, the evolutionary algorithms and their variants are introduced. Chapter 3 describes CUDA programming in order to design efficient implementation. Proposed implementation and design decisions are given in Chapter 4. In Chapter 5.1, set of problems is presented for the purpose of test, comparison, and benchmark of the implementation. This is followed by the evaluation of the proposed implementation and its comparison to the standard methods. Finally, conclusion and main ideas are given in Chapter 6.

2. Evolutionary Algorithms

Optimization. What better word would describe the ultimate goal of the people, our civilization, and maybe even the Universe. With a bit of exaggeration, the optimization is the cause of all ramifications we can see around. What else are the laws of physics than minimization processes, our pursuit of knowledge and self–improvement than maximization of self-worth, industrial revolutions than minimization of human labor, maximization of goods production and standards of living at the same time. In my opinion, the optimization is the reason that drives us forward, makes us better, and guides our decisions.

One of the most extraordinary optimization processes is, without a doubt, evolution. Evolution is nothing less than Nature way of improving species to better adapt to their environment. The father of the evolution idea is undoubtedly Charles Darwin, who first came up with an idea of evolution in his work "On the origin of species" [**?**]. According to him, the individuals better adapted to the environment have a higher potential to reproduce and pass their genetic material — including the predisposition to survive in the environment — to their offspring. This is sometimes referred to as "Survival of the fittest".

These exact ideas inspired scientists and formed the field of Evolutionary Algorithms (EA).[1] EA are stochastic, population-based optimization algorithms inspired by the biological evolution. In general, a set of individuals called the population compete with each other for a place in the next generation. Each individual represents one possible solution to the problem at hand. The quality of the individual is proportional to the quality of solution it represents. This quality is usually expressed using the fitness (or sometimes called objective) function. As with natural evolution, the probability of survival is proportional to the solution quality – the better the solution is, the higher the probability of survival [**?**]. In the EA, the process of producing new generation is usually called selection.

Evolution is not only about survival but also about reproduction. Evolutionary algorithms took inspiration here as well and use special operators for each generation – usually called evolution operators or variation operators. As the name suggests, these operators make sure the population explores new solutions. The two most common operators are crossover and mutation. During crossover, individuals from the current population (usually called parents) produce new individuals (usually called offspring) by combining their genetic information. In some way, crossover mimics reproduction in the Nature, during which the DNA of offspring is by majority determined by its parents DNA. Mutation, on the other hand, causes a random minor alteration of the individual genetic material and is able to introduce novel patterns that did not previously occur in the population [**?**].

In general, EA repeat the steps above until a sufficiently good solution is found or until a maximum number of generations is reached. The general pseudocode of the EA algorithm is presented in Algorithm 2.1. I will present more accurate implementations of various algorithms throughout this chapter.

[1] I will refer to all the population–based algorithms in this work as Evolutionary Algorithms. These include, among others, Genetic Algorithms, Evolution Strategies, and Particle Swarm Optimization.

Algorithm 2.1: General Evolution Algorithm

Result: evolved population
initialize population;
repeat
    reproduction;
    mutation;
    evaluate population;
    selection;
until *termination criteria met*;

2.1 Terminology

This section will introduce the necessary terminology that I am going to use throughout this work. I took most of the terminology from **?**.

Problem is the function we want to solve. It can be both a satisfactory (we are looking for any solution) or optimization (we are looking for the best possible solution with respect to some measure) problem.

Genotype is encoding of the solution. It is an equivalent to the DNA – it compresses all the data needed to produce solution.

Gene is one part of the genotype. It may be one bit if the genotype is encoded as a binary string, or scalar if the genotype is a vector or tensor.

Representation or genotype space is the space of genotype.

Phenotype is the solution built from genotype. Genotype and phenotype can be equivalent – for example, a real function optimization problem may have a genotype vector of real numbers representing one of the solutions. Generally, however, genotype and phenotype can be different [**?**].

Search space is the space of phenotype.

Objective function is the function the algorithm optimizes. For the previous example of real function optimization, the function itself can be the objective function. For satisfactory problems may be objective functions just mapping from the genotype to the zero (problem unsatisfied) or one (problem satisfied). I will denote the objective function as f_o.

Fitness function is the measure of solution quality. Note that the domain of the function is the search space. Unlike the objective function, the fitness function should guide the algorithm during the optimization. It can use the objective function directly, but there are other options available. For example, if the goal is to maximize objective function $f_o(x)$, but the algorithm is written such that it minimizes the fitness function, one may specify the fitness function as $f_f(x) = \frac{1}{f_o(x)}$. I will denote the fitness function as f_f.

Fitness value is the value of fitness function of a given genotype.

Evaluation is the process of obtaining fitness value.

Individual is a genotype together with its fitness value. I will denote an indi-

vidual by $\boldsymbol{p}$ and his fitness as $f_f(\boldsymbol{p})$.

Population is a multiset of individuals – it is possible to have identical genotype in the population multiple times. I will still distinguish different individuals with the same genotype.

Initialization is the process of creating the first population.

Exploitation is the effort to use knowledge from the history in order to maximize the expected outcome [**?**].

Exploration is the effort to discover new rules about the problem at hand, although the expected outcome does not need to be the best possible. In general, the biggest problem in EA is the right balance between exploitation and exploration. Emphasizing the exploitation may cause getting stuck in the local minima, whereas focusing on the exploration may cause the algorithm never to converge. Note that EA is not the only field suffering from this, but for example Reinforcement Learning deals with the same problem [**?**].

Evolution operators are individual steps performed in each iteration.

Selection is an evolution operator that draws individuals from the current population to the next one. The selection should consider the fitness value of each individual in the population and pick them proportionally. Selection, therefore, implements the "Survival of the fittest" law. From the mathematical point of view, selection task is to move the population into the more promising area of the search space and possibly reduce the variance of the population [**?**]. In other words, selection is the exploitation step in the algorithm.

Crossover is evolution operator, that combines some individuals (usually called *parents*) in order to create one or more new individuals (usually called *offspring*). Offspring are in most cases a combination of their parents, similarly to how DNA of the child is a combination of DNA of its parents.

Mutation is an evolution operator that modifies one individual. Mutation can be based on randomness (for example replacement of value in the genotype by a different value) or specialized for the problem at hand. This time the example can be Traveling Salesman Problem, where one possible mutation operator can "untwist" paths that cross each other. This mutation is, in fact, a local optimization technique because the mutated way is necessarily shorter [**?**].

Variation operators are operators that change the variation of the population but not the mean. Variation operators are the exploration steps in the algorithm and should balance the exploitation strength of the selection. Both crossover and mutation are variation operators [**?**].

Memetic operator is an operator that uses local optimization in every iteration. Alternatively, "memetic algorithms use local optimizer to every solution before evaluation" [**?**]. One such example is the mutation operator for Traveling Salesman Problem problem mentioned earlier. Another example is the training of ANN. The memetic operator would be one run of the backpropagation algorithm in each generation.

(a) One–point crossover (b) Bit–flip mutation

Figure 2.1: Genetic Algorithms operators

2.2 Genetic Algorithms

Genetic Algorithms (GA) are probably the simplest Evolutionary Algorithms (EA) out there and were introduced by **?**. In the GA, individuals are binary string of length n, formally

$$\boldsymbol{p} \in \{0,1\}^n$$

Fitness function maps genotype into real values

$$f_f : \{0,1\}^n \to \mathbb{R}$$

The typical crossover operator is *one–point crossover*. This type of crossover requires two parents and produces two offsprings. As the first step, the algorithm uniformly samples a random integer s in the range $[1, n-1]$. The first offspring will receive genetic material up to index s from the first parent and the rest from the second one. The second offspring is created the same way, except the roles of parents are exchanged. An example for $n = 10$ is in Figure 2.1a. In order not to modify all the individuals in the population, there is only $p_c \in [0,1]$ probability that the individual undergos the crossover, and the children replace their parents [**?**].

The mutation operator for GA is in most cases the bit–flip mutation. During this mutation, each gene mutates with probability p_m. One example of mutation is in Figure 2.1b. Mutation has two objectives – it makes sure the algorithm is not trapped in local optima and it sustains genetic diversity in the population. Mutation serves as a secondary search operator [**?**].

The difference between crossover and mutation is such that crossover is rather "search" exploitation technique. It combines individuals from the current population and exploits them to find possibly better individuals. On the other hand, the mutation is a more local search technique, and based on the probability p_m, it searches over the whole representation space.

Finally, the selection stage takes place. There are various selection techniques, and in this particular case, I will use tournament selection. Tournament selection compares fitness values of two random individuals and places the better one into new population. This repeats as many times as needed to form a new generation with desired number of individuals.

For the cases where the population size does not change between generations, there is a high probability that the same individual will be in the following population multiple times. Nonetheless, because variation operators (crossover and mutation) keep diversity in the population, duplicities in the population are not a problem.

The pseudocode of the simple genetic algorithm described above is depicted in Algorithm 2.2. Firstly, the population is randomly initialized, and then it

Algorithm 2.2: Simple genetic algorithm

Input: d problem dimension, l population size, g generations, f_f, p_c, p_m
Result: evolved population
population ← randomly initialized;
foreach *gen in* $0..g$ **do**
  foreach *individual in* $0..l$ **do**
    if $rand() < p_c$ **then**
      One point crossover with random individual
    if $rand() < p_m$ **then**
      Bit–flip mutation
  Evaluate population using f_f;
  population ← pick up l individuals using tournament;
return *population*

(a) Two–point crossover (b) Uniform crossover

Figure 2.2: Advanced crossover operators

undergoes crossover, mutation, evaluation, and selection in the loop. Finally, the evolved population is returned from the algorithm.

I will refer to the algorithm described above as "Simple Genetic Algorithm". In reality, scientists come up with various operators that can improve convergence or can help in specific types of problems. Through the following paragraphs, I will focus on these techniques, inspired mainly by the book of authors **?**.

2.2.1 Advanced crossover operators

The simple genetic algorithm uses one–point crossover. It is straightforward to extend it into *two–points crossover* where each genome is split into three parts. Each offspring then receives the first and third part of the genome from one parent and the middle one from the second one. An example of a two–points crossover is in Figure 2.2a.

Sometimes, the crossover operator is generalized more and forms a k–point crossover. The genotype partitions into k parts, and these parts are interleaved in the offspring. A special case is an *uniform crossover* — the offspring are constructed in such a way that they are a uniform combination of their parents. Each gene in the offspring has an equal probability of whether it will be received from the first or second parent. The second offspring will follow the same pattern except with parents swapped. An example of the uniform crossover is in Figure 2.2b.

I have described the crossover operators such that the offspring replace their parents. That is a popular design decision, however, one does not need to enforce it and may decide to create offspring in addition to the parents. To ensure the

population does not increase in size over the generations, the selection operator can be implemented in a way that a specified number of individuals will be drawn up. I will discuss this possibility in Chapter 2.3.

2.2.2 Selection operators

The selection operator is one of the most important one. It is the only operator working with fitness values; therefore, the operator is responsible for the exploitation part of the algorithm. The simple genetic algorithm presented in Algorithm 2.2 used tournament selection — I will analyze other types of selections through the following paragraphs.

Selection operators are equivalent to Darwin's selection process in the Nature. It allows to preserve perspective genetic traits and discard the bad ones. That corresponds to the "Survival of the fittest" idea. The distinction between good and bad individuals is with respect to the fitness function; selection operators, therefore, favor individuals with better fitness value.

The degree into which operators endorse better individuals is called *selection pressure*. Selection pressure is a primary indicator of the algorithm and needs to be balanced with other variation operators. Too immense selection pressure may cause the algorithm to converge prematurely. On the other hand, too weak selection pressure may prolong the convergence, and the algorithm may need unfeasible amount of steps before convergence.

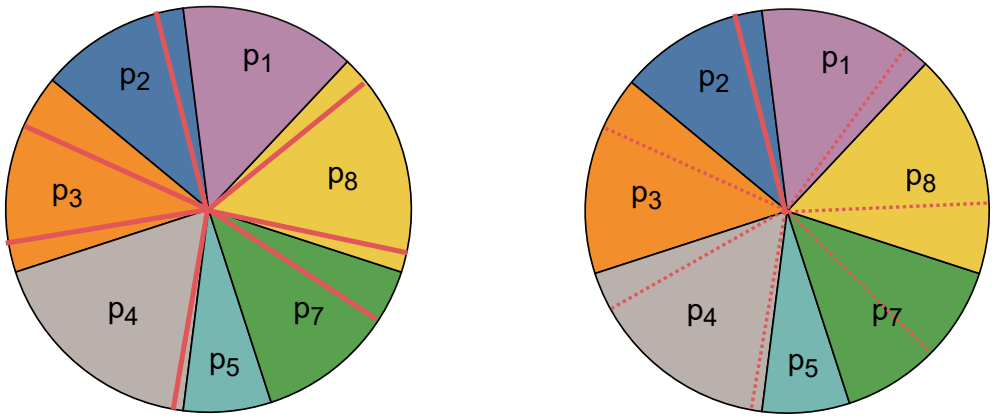
Tournament selection is one of the most popular selection operators because it is easy to implement and the absolute values of fitness are not relevant (rather relative order of the individuals is what matter). Tournament selection, therefore, keeps the selection pressure constant. We may extend it naturally to k parents, it increases the drawn probability for better individuals and thus increases the operator selection pressure.

Another popular operator is the *Roulette Wheel selection*. In this operator the probability of selection is equal to the absolute value of the fitness.

$$P\left[\boldsymbol{p}_i\right] = \frac{f_f(\boldsymbol{p}_i)}{\sum_{j=0}^{l} f_f(\boldsymbol{p}_j)}$$

This can be imagined as a roulette wheel, where individuals represent a sector of a wheel. The size of the sector is proportional to the fitness value of the individual. The selection procedure simply spins the roulette n–times and selects the individual on which the roulette stopped (for population size equal n). An example of the roulette wheel selection is in Figure 2.3a. Selection operators that depend on the absolute values of fitness value are usually called proportional–based selection operators.

A slight improvement of the roulette wheel selection is *Stochastic Universal Sampling* (SUS). Because roulette wheel selection samples individuals independently, it may select the same individuals more time than would be expected given its sector size. In the example in Figure 2.3a, the expected number of draws for individual p_3 is one. However, it has been drawn twice. Stochastic Universal Sampling handles this issue by sampling only one number in the range $\left[0, \frac{\sum_i f_f(p_i)}{n}\right]$ and drawing each new individual with $\frac{\sum_i f_f(p_i)}{n}$ offset (where n is



(a) Roulette wheel selection. Red lines represent a sampled individuals.

(b) Stochastic universal sampling. Red line represents a sampled number and dotted lines are sampled individuals with the 1/7 offset.

Figure 2.3: Roulette based selection

population size). The SUS is unbiased and the drawn distribution is closer to the fitness values distribution in contrast to the roulette wheel selection. An example of SUS is in Figure 2.3b.

Notice that for both selection operators individuals with the higher fitness value have a higher probability of being picked up to the successive population. These operators are therefore intended for maximization problems. Also, because the fitness value directly affects the sector size, the fitness function must be positive over the whole domain. Individuals with negative fitness may be discarded; however, then the whole selection procedure would lose relevance.

One additional drawback of both roulette wheel selection and SUS is that the probabilities depend on fitness absolute values. Let us look at two cases:

- The population fitness values are distributed in the range $[99, 100]$.
- The population fitness values are distributed in the range $[1, 2]$.

In both cases, the expected difference of fitness between any two individuals is the same. Nevertheless, because roulette–based selection takes into account absolute values of fitness, the probability of drawing the best individual in the first case is smaller compared to the second case (10% versus 13% for population with 10 individuals). This is a side effect of proportional–based selection. It is even more perceptible if few individuals have extremely high fitness and span almost the whole roulette.

The drawback of proportional–based selection is that it does not create enough selection pressure when the population has roughly the same fitness value. This problem occurs at the end of the evolution because all the individuals are optimized over the generations and usually have very similar fitness values. Note that tournament selection does not suffer from this, as only the order of the individuals determines the selection pressure.

One way to eliminate this problem is to rescale fitness to have more desired properties. One common example is linear scaling into interval $[l, u]$ so that the

Fitness value	Rank	New fitness for $[1, 2]$	Selection probability for $[1, 2]$	New fitness for $[1, 10]$	Selection probability for $[1, 10]$
12	1	2	27%	10	38%
7	2	1.75	23%	7.5	29%
1	3	1.5	20%	5	19%
0	4	1.25	17%	2.5	10%
-5	5	2	13%	1	4%

Table 2.1: Rank selection pressure

individual with $\min f_f(x_i)$, respectively $\max f_f(x_i)$, has new fitness equal to l, respectively u; and the rest of them are scaled linearly between them.

$$f_{scale}(\boldsymbol{p}) = \frac{f_f(\boldsymbol{p}) - \min\limits_i f_f(\boldsymbol{p}_i)}{\max\limits_i f_f(\boldsymbol{p}_i)} \cdot (u - l) + l$$

Alternatively, logarithmic and exponential scale function is possible, so there will be more, respectively, less emphasis on individuals from the beginning of the sorted sequence.

Special example of fitness scaling is the *rank selection*. To eliminate dependency on the absolute value of the fitness function, rank selection assigns artificial fitness values to the individuals based on their order. One common rank evaluation function is linear — it assigns new fitness values in the interval $[l, u]$ such that the best individual has new fitness value l, the worst one u, and the rest of the population is linearly distributed in this interval [**?**] (I expect minimization problem). Different scaling functions are possible as well.

The advantage of rank selection is a steady selection pressure, and unlike mentioned roulette wheel selection, it can handle fitness functions with negative values. Moreover, it is easy to control the selection pressure using interval size, as shown in Table 2.1. It also does not suffer from outliers — individuals with very high or low fitness in comparison to the rest of the population. Rank selection can avoid premature convergence, but computation costs can be very high because the population needs to be sorted, fitness values reevaluated, and another proportional–based selection operator needs to follow.

Selection operators may use traditional approaches from search algorithms, for example *simulated annealing*. Simulated annealing is an analogy to metallurgy, whereby precise control of heating and cooling of material improves its physical and chemical properties. In general, it accepts worse solutions as long as the temperature τ is high enough. The temperature τ starts high, so almost all the solutions are accepted from the beginning. The temperature slowly decreases over the generations down to zero. The decrease is usually exponential: $\tau_g = \tau_0(1 - \alpha)^{1+\frac{\beta \cdot g}{G}}$ where g is the current generation, G maximum number of generations, $\alpha \in [0, 1]$, $\beta \in \mathbb{R}^+$, and $\tau_0 \in \mathbb{R}^+$ are parameters.

One example of simulated annealing is the Boltzmann selection operator. It draws individuals with probability $P \propto exp\left(-\frac{f_{max} - f_f(\boldsymbol{p})}{\tau}\right)$ where f_{max} is the best fitness in the population and τ is the current temperature (known as Boltzmann probability). Note that the Boltzmann selection is once again a proportional–

based selection operator and may suffer from the same problems as the roulette wheel selection.

The last selection operator I would like to discuss is *elitism*. For some problems may be beneficial to preserve the best individuals from the population. This is especially helpful for problems where the algorithm can find the optimal solution using small local steps. Mutation and crossover may destroy the best individual in the population, and it may be difficult for the algorithm to find it once again if its genetic material has not been propagated yet.

The elitism mechanism is an operator which makes sure the best individuals endure in the population. It is usually implemented in a way it copies the k best individuals from the population before all other operators execute, and copy them back afterwards. Alternatively, it prohibits their change or appends them to the population afterward. Note that elitism is not truly a selection operator, but rather an insurance mechanism not to lose the best individuals in the population. Traditional selection operator still needs to be present in order for the algorithm to work.

2.3 Real–Coded Evolutionary Algorithms

Real–coded evolutionary algorithms are natural extension of GA by expanding genotype into real numbers.

$$\boldsymbol{p} \in \mathbb{R}^n$$

Alternatively, the relaxed definition allows genotype of an arbitrary number of dimensions. That may be helpful for problems, which expect a particular structure of the solution – for example image processing filters [**?**]. Note that using the different genotype structure disallows direct use of some operators, like one– and two–point crossovers, because the split point is ambiguous. On the other hand, some operators, like the uniform crossover, can easily handle multidimensional genotype without much change.

Origin of real–coded evolutionary algorithms, more precisely Evolution Strategies (ES), dates back to 1970s at Technische Universität Berlin, where **?** presented ES for the first time [**?**]. Originally, ES have not been used as an optimization algorithm of real functions, but "rather as a set of rules for the automatic design and analysis of consecutive experiments with stepwise variable adjustement driving a suitable object/system into the optimal state in spite of environmental noise" [**?**]. It turned out that this simple stochastic search technique outperformed traditional–based methods. Although there is no guarantee ES will find the optimal solution, they are in most cases able to find a near–optimal and sufficiently good enough solution.

This section takes inspiration mainly from the work of **?** and **?**. Moreover, some techniques originated in ES are applicable for general real–coded evolutionary algorithms as well. Therefore, I will not make rigorous distinction between real–coded evolutionary algorithms and ES through this section.

Real–coded evolutionary algorithms can use the same selection and crossover operators as GA, although it is not recommended **?**. In general, good crossover operators should

- preserve population mean because crossover operators do not work with a fitness function and therefore should not introduce bias in the search space,
- slightly increase the variance of the population, to balance the pressure of selection operators, that usually decrease the variance of the population,
- increase variety of the population, so the algorithm effectively searches neighborhood solutions,
- allow reachability of any solution in the search space.

Crossover operators from GA may fulfill these requirements, however, real–coded encoding allows a more diverse set of operators. I will describe them a bit later.

ES also introduced different selection schemes:

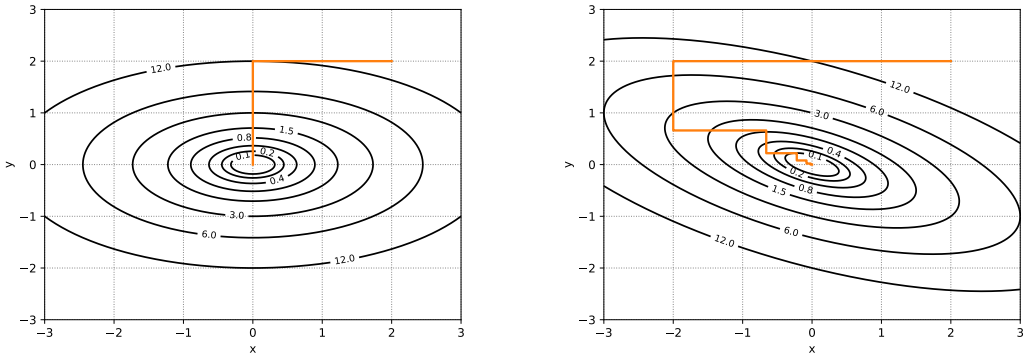
- $(\mu + \lambda)$ evolution strategy, also known as *plus schema*,
- (μ, λ) evolution strategy, also known as *comma schema*,
- steady–state evolution strategy.

Here, the μ represents a number of parents and λ number of offspring. In the first two cases, μ individuals are picked to the next generation. In the plus schema, these individuals are taken from both parents and offspring. For the comma schema, μ individuals are drawn only from the offspring and parents are discarded. Obviously, to have at least μ distinct individuals in the next generation, it must hold $\lambda \geq \mu$. For stochastic selection operators, where it is possible to draw the same individual multiple times, it may be beneficial to have more offspring than parents.

Both plus and comma selection schemes generate μ offspring and resample the whole population for the next generation. On the other hand, steady–state evolution strategy generates just one (or a very few) individuals in each generation and tries to reintegrate them back into the population. The steps of the algorithm are:

1. Generate a new individual and evaluate it using the fitness function.
2. Choose an individual from the population that the new one can replace.
3. Decide whether the new individual should replace the old one.

Step 2 is known as the replacement strategy, and some common variants are to replace the oldest, worst, or random individual. Step 3 is known as the replacement condition. The most common variant is to replace the old individual only if the new one is better; however, it is also possible to use simulated annealing. The steady–state expresses the idea that the population undergoes only a tiny change in each generation [**?**].



(a) Separable ellipsoid function.

(b) Non–separable ellipsoid function.

Figure 2.4: Difference between separable and non–separable function for optimization. Trajectories of hill climbing algorithm are depicted in orange.

2.3.1 EA crossovers

Encoding genotype as a vector of real numbers allows, except crossover operators from GA, extra crossover operators working with real numbers. One argument may be to exploit correlation within the population.

Let us take for example two functions:

$$f_{sep}(x, y) = x^2 + 3y^2$$
$$f_{nosep}(x, y) = x^2 + 3y^2 + 2xy$$

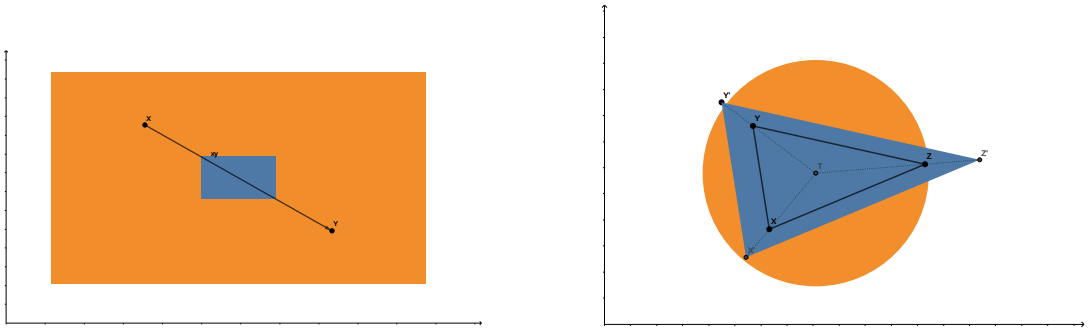
These functions are shown in Figure 2.4. Running simple hill climber algorithm [**?**], one may get trajectories displayed in the figure. Notice that the trajectory of the function f_{nosep} is much longer – it took 200 steps to reach global minimum for the function f_{sep}, however up to 400 steps for the function f_{nosep} (the step size was equal 0.02 for both runs). The first function is separable, it can be written in the form $f_{sep}(x, y) = f(x) + f(y)$, which causes this behavior. Hill climbing algorithm can easily find optimum because $\min f_{sep}(x, y) = \min f(x) + \min f(y)$, therefore, it can search for the optimum for each variable and then aggregate the results. That is not true for the function f_{nosep}, because it is not separable, and using the local search technique is not as straightforward. Separable functions are more challenging and harder in some sense, exactly the kind of problems EA should solve efficiently.

The problem of crossover operators from GA is precisely the same as I have just shown. These operators work on the genes independently and do not use knowledge about the correlation of the genotype space within the population. It is therefore necessary to use different crossover operators tailored for the real–coded EA.

One of the simplest real–coded EA crossover is *arithmetic crossover* (also known as a convex combination). For parents $\mathbf{p_1}$ and $\mathbf{p_2}$, the offspring $\mathbf{o}$ is created using the following formula.

$$\mathbf{o} = \alpha \mathbf{p_1} + (1 - \alpha)\, \mathbf{p_2}$$

where $\alpha \sim U(0, 1)$ is an uniformly distributed variable in interval $[0, 1]$. It is possible to generate second offspring by setting $\beta = 1 - \alpha$ and substitute β



(a) Blend crossover sample space with $\alpha = 0.5$ (orange) and $\alpha = -0.3$ (blue) in two dimensions.

(b) Simplex crossover example in two dimensions. Sampling space of simplex (blue) and hypersphere (orange).

Figure 2.5: Real–coded crossover operators

for α. This operator uses the same α for every dimension of the vector, so it is sometimes called *whole arithmetic crossover*. An easy extension is to use $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ vector, where $\alpha_i \sim U(0, 1)$ and create offspring using similar formula.

$$\mathbf{o} = \boldsymbol{\alpha} \cdot \mathbf{p_1} + (1 - \boldsymbol{\alpha}) \cdot \mathbf{p_2}$$

where $\cdot$ is an element–wise vector multiplication. Further extension of the algorithm is to use up to k parents $\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_k}$ and create offspring by their linear combination.

$$\mathbf{o} = \boldsymbol{\alpha}_1 \cdot \mathbf{p_1} + \boldsymbol{\alpha}_2 \cdot \mathbf{p_2} + \cdots + \boldsymbol{\alpha}_k \cdot \mathbf{p_k}$$

where $\sum_{i=1}^{k} \boldsymbol{\alpha}_i = \mathbf{1}$.

Arithmetic crossover is just a linear combination of parents, it does not satisfy the condition on Page 13 to "increase variety of the population". Authors **?** address this issue in *blend crossover* operator. For parents $\mathbf{x}$ and $\mathbf{y}$, the genes of offspring $\mathbf{o}$ are sampled randomly from uniform distribution specified by its parents.

$$o_i = U\left(x_i - \alpha\left(y_i - x_i\right), y_i + \alpha\left(y_i - x_i\right)\right)$$

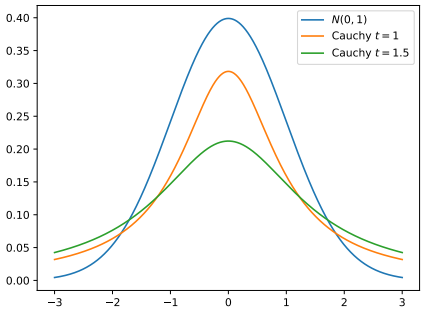
Here, α is user specified parameter and the algorithm is sometimes called BLX–α crossover. An example of BLX is in Figure 2.5a.

Another type of crossover operator is *simplex crossover*. It uses simplex to sample offspring. Simplex is a generalization of triangle into more dimensions. It is defined in n-dimensional space using n+1 points. For two-dimensional space the simplex is a triangle, for three-dimensional space the simplex is a tetrahedron, and so on. From the definition, the simplex crossover require $n + 1$ parents $\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}, \mathbf{p_{n+1}}$. It calculates their centroid as:

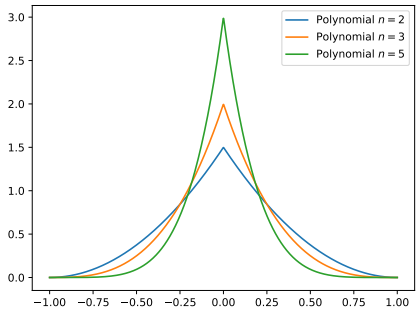
$$\mathbf{k} = \frac{\sum_{i=1}^{n+1} \mathbf{p_i}}{n + 1}$$

and then expand the simplex by ε along each of the direction $\mathbf{p_k} - \mathbf{k}$. As a result, this new, expanded simplex is defined by the points

$$\mathbf{p_i'} = \mathbf{p_i} + \varepsilon\left(\mathbf{p_i} - \mathbf{k}\right)$$



(a) Normal and Cauchy distributions



(b) Polynomial distributions

Figure 2.6: Mutation distributions

The algorithm then samples the offspring from this simplex. An example of a simplex crossover in two-dimensional space is in Figure 2.5b with $\varepsilon = 1.5$. I would argue the hypersphere sample space can be used as well, as is shown in Figure 2.5b with $\varepsilon = 1.2$.

2.3.2 Real–coded mutation operators

An equivalent of bit–flip mutation in GA is *uniform mutation*. With small probability p_m it replaces gene by a random value sampled from $U(l, u)$, where interval $[l, u]$ is the scalar domain. This operator basically tries random numbers and can rarely lead to a good solution. Moreover, it does not exploit the distribution of the population.

More successful and frequently used is *normal mutation*. It performs a little change of the individual by adding sample from a normal distribution with zero mean.

$$p_i = p_i + N(0, \sigma)$$

The σ is the deviation of the normal distribution. It is parameter set up by the user and is usually kept low. Notice that normal mutation does not use mutation probability p_m. Because the σ is usually small and the distribution has zero mean, the changes do not have significant impact on the population.

Normal mutation can be further extended to perform well on more complicated problems. Instead of the σ variable, it is possible to introduce $\boldsymbol{\sigma}$ vector, which allows different deviation in each dimension.

$$p_i = p_i + N(0, \sigma_i)$$

To address non–separable functions, we may need the whole covariance matrix Σ for successful adaptation. Normal mutation with covariance matrix is sometimes called *correlated mutation*. Notice that in order to use correlated mutation in n dimensions, the algorithm needs $\frac{n(n+1)}{2}$ parameters set up by the user just for the correlation matrix. Correlated mutation has, therefore, just a trivial practical usage on its own. Other algorithms address this issue by setting these parameters based on the observations — the Covariance Matrix Adaptation – Evolution Strategy (CMA–ES) is the most known of them.

The normal distribution is probably the most used mutation distribution, but different zero–mean distribution can replace it. **?** used Cauchy distribution to fasten the search. As shown in Figure 2.6a, Cauchy distribution does not decrease as fast as normal distribution and, therefore, searches wider genotype space. On the other hand, if we desire to shrink the searche space, we may use polynomial distribution, which probability density function is defined as

$$p(x) = 0.5\,(n+1)\,(1-|x|)^n$$

where n is user–defined parameter and function domain is interval $[-1, 1]$. An example of polynomial distribution is in Figure 2.6b.

2.3.3 Adaptive operators

It is usually helpful to alter algorithm parameters through evolution to better adjust to the problem at hand. For example, one may decrease the deviation of the distribution as the population is converging to the optima. *Adaptive operators* is a general term for operators, which adjust their parameters based on the algorithm progress.

Decay operators decrease their value based on the generation number. It may decrease the mutation or crossover probability [**?**], mutation strength, population size, and many more. The central idea behind is that the changes are getting smaller as the algorithm proceeds, and the algorithm has a better chance to converge.

One of the most straightforward decay function is linear, which decrease the parameter value from u to l linearly.

$$decay_l(u, l) = u - \frac{g}{G} \cdot (u - l)$$

where g is the current generation and G is total number of generations. Another popular decay function is exponential:

$$decay_e(s, r) = s \cdot r^g$$

where s is a starting value of the parameter and $r \in [0, 1]$ is a decay rate. Note that in order to get the same decay rate for a different number of generations, the rate must be reevaluated. The final decay rate I would like to mention is polynomial decay.

$$decay_p(s, p) = s \cdot \left(1 - \frac{g}{G}\right)^p$$

where s is a starting value and p is an adjustable parameter. Examples of decays beginning at value 5 and taking 400 generations are shown in Figure 2.7.

Another popular approach is the *one–fifth rule* [**?**]. The main idea behind it is to increase mutation deviation (usually called step–size) after successful mutation and decrease it otherwise. The reasoning is that if too many individuals are better, then the search is probably too local, and by increasing the step size, we enlarge the search domain. Consequently, a few successful individuals indicate that the step is too big and should decrease.

As a rule, authors **?** suggested 1/5 success rate as the threshold value. If at least 20% offspring are better, then the step–size should be increased by a factor

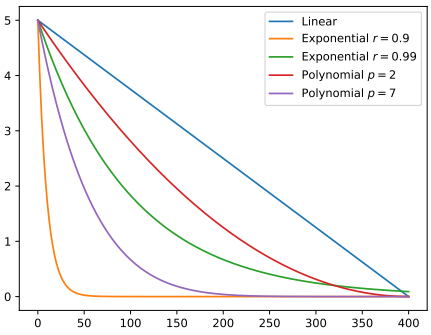


Figure 2.7: Decay rates

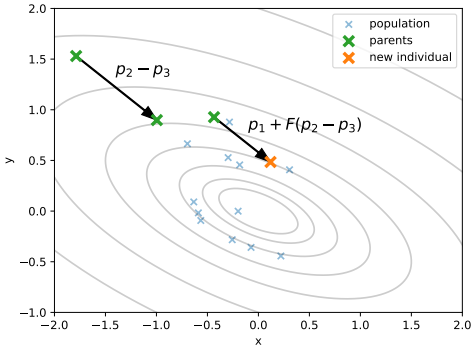


Figure 2.8: Differential evolution crossover with $F = 0.7$

of 1.5 and decreased by a factor of $1.5^{-1/4}$ otherwise [**?**]. However, the proof of this rule by **?** was done only for unimodal function and small populations. It is generally not a good balance for more complex problems. Better results can be achieved using more advanced techniques to adapt parameters, such as CMA–ES.

2.3.4 Differential Evolution

Differential Evolution (DE) is a kind of real–coded evolutionary algorithm, suggested by **?**. DE applies special operation to the population, that combines crossover, mutation, and selection. DE is sometimes considered as a directional–based search technique, because it takes into account distribution of the individuals in the genotype space. It can therefore solve non–separable functions better in comparison to traditional real–coded evolutionary algorithms.

DE uses four distinct parents $\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}, \mathbf{p_4}$. Firstly, it constructs trial vector $\mathbf{v}$ such that

$$\mathbf{v} = \mathbf{p_1} + F\left(\mathbf{p_2} - \mathbf{p_3}\right)$$

where F is an user–defined positive parameter controlling the strength of the direction. The vector $\mathbf{p_2} - \mathbf{p_3}$ is the direction from parent $\mathbf{p_3}$ to parent $\mathbf{p_2}$ and reflects the correlation within the population. One example is in Figure 2.8. Notice how a newly created individual is much closer to the optimum compared to its parents.

There is a possibility to use more parents and combine them. Moreover, there is option not to sample parents uniformly but substitute the best individual in the population as either $\mathbf{p_1}$ or $\mathbf{p_2}$. Alternatively, it is possible to use the same individual as the source of the direction, as well as the starting position. Expressed mathematically by substitution $\mathbf{p_1} = \mathbf{p_3}$. The formula of the crossover with five parents is

$$\mathbf{v} = \mathbf{p_1} + F_1\left(\mathbf{p_2} - \mathbf{p_3}\right) + F_2\left(\mathbf{p_4} - \mathbf{p_5}\right)$$

Last but not least, this crossover operator can be easily used within traditional real–coded evolutionary algorithms.

After the crossover, DE performs the uniform crossover between the trial vector $\mathbf{v}$ and parent $\mathbf{p_4}$ to finally create offspring $\mathbf{o}$. The mutation rate is controlled

by the parameter $C \in [0, 1]$ that represents a probability that offspring $\mathbf{o}$ inherits gene from the trial vector $\mathbf{v}$. Finally, the offspring $\mathbf{o}$ is compared to the original parent $\mathbf{p_4}$ and replaces it, if it is better. This step is equivalent to selection and DE is a kind of steady–state evolution strategy, as defined on Page 13.

2.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another population–based stochastic optimization technique inspired by a flocking of birds and schooling of fish in the Nature. It was introduced by **?**. The PSO is somehow similar to the real–coded evolutionary algorithms and ES in the sense that the individual is a vector of real numbers. In the PSO terminology, the individuals are called *particles*.

Through the time, the PSO went over plenty of improvements. Two major versions, namely Standard Particle Swarm Optimization 2006 (SPSO–2006) and Standard Particle Swarm Optimization 2011 (SPSO–2011), were released as a baseline for further improvements. I will focus on these versions as specified by **?**.

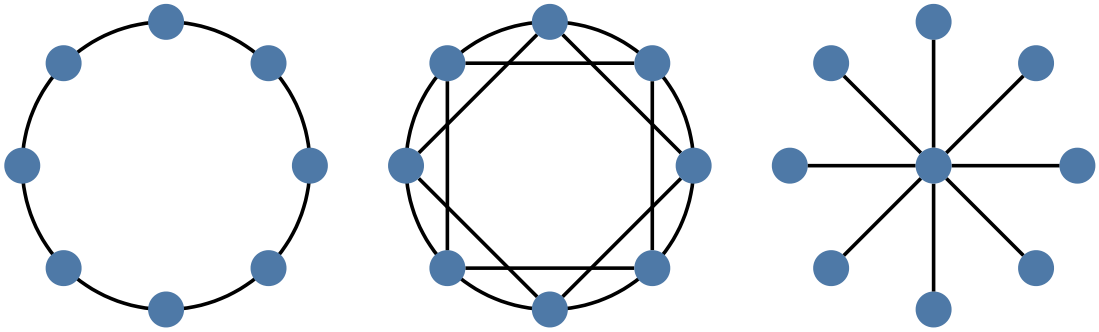
Except for the position, each particle $\mathbf{p_i}$ has a velocity vector $\mathbf{v_i}$, which defines the particle direction and speed. The velocity vector is then used for a position update. Unlike real–coded evolutionary algorithms, modifications to the particles are not made directly to their positions, but rather by manipulating with the velocity vector. These two components are the reason why PSO got its name.

In addition to velocity, each particle $\mathbf{p_i}$ remembers its own best position, usually called *local best* and denoted by $\mathbf{L_i}$. This variable is updated after every step if the particle fitness value is better than the fitness value of $\mathbf{L_i}$.

Similarly, each particle remembers the best position it received from the swarm, usually called *global best* and denoted by $\mathbf{G_i}$. After each step, particle

Algorithm 2.3: General Particle Swarm Optimization (PSO) algorithm

Input: s population size, *steps* number of steps, f_f fitness function
foreach *particle* i *in* $1..s$ **do**
  Initialize particle position $\mathbf{p_i}$;
  Initialize particle velocity $\mathbf{v_i}$;
  Initialize local and best positions $\mathbf{L_i} = \mathbf{G_i} = \mathbf{p_i}$;
foreach *step in* $0..steps$ **do**
  foreach *particle* i *in* $1..s$ **do**
    Update velocity $\mathbf{v_i}$ based on $\mathbf{p_i}$, $\mathbf{L_i}$, and $\mathbf{G_i}$;
    Update position $\mathbf{p_i}$ based on $\mathbf{v_i}$;
    if $f_f(\mathbf{p_i})$ *is better than* $f_f(\mathbf{L_i})$ **then**
      $\mathbf{L_i} \leftarrow \mathbf{p_i}$;
    foreach *particle* j *in* $Neighborhood(i)$ **do**
      if $f_f(\mathbf{p_i})$ *is better than* $f_f(\mathbf{G_j})$ **then**
        $\mathbf{G_j} \leftarrow \mathbf{p_i}$;
return *particle* $\mathbf{p_k}$ *such that* $k = \arg\min_i f_f(\mathbf{p_i})$



(a) Ring topology (b) Dense ring topology (c) Star topology

Figure 2.9: PSO neighborhood topologies

informs surrounding particles about its position and fitness value. Each particle may inform a different set of particles. Which particles to inform is defined by particle *neighborhood*, and I will focus on it a bit later. Each particle then accumulates information from its surrounding (the information it received from other particles) and may update its global best position $\mathbf{G_i}$.

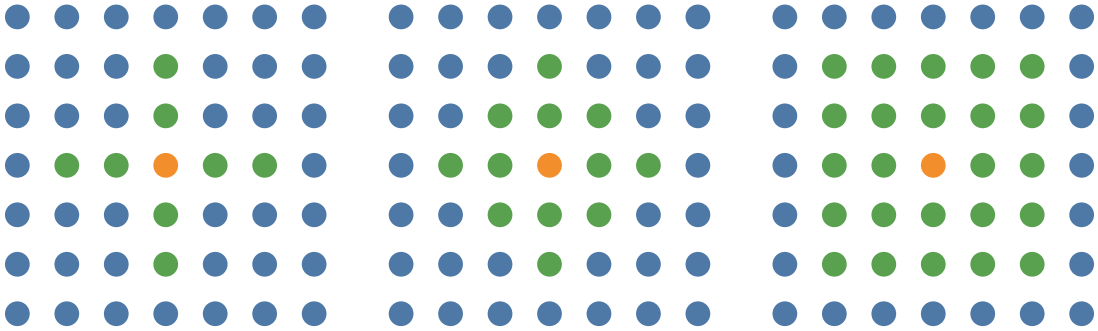
The general PSO implementation is in Algorithm 2.3. The difference between SPSO–2006 and SPSO–2011 is in the way they update the velocity $\mathbf{v_i}$.

2.4.1 Neighborhood

Neighborhood, also called topology, specifies interconnections between particles and how the particles communicate their best positions in the swarm. It directly controls information flow and speed of convergence. If the neighborhood is too large all the particles soon receive the same best position and will attempt to reach it without much exploring. If the neighborhood is too small, particles may focus too much on the exploration because the global information is missing.

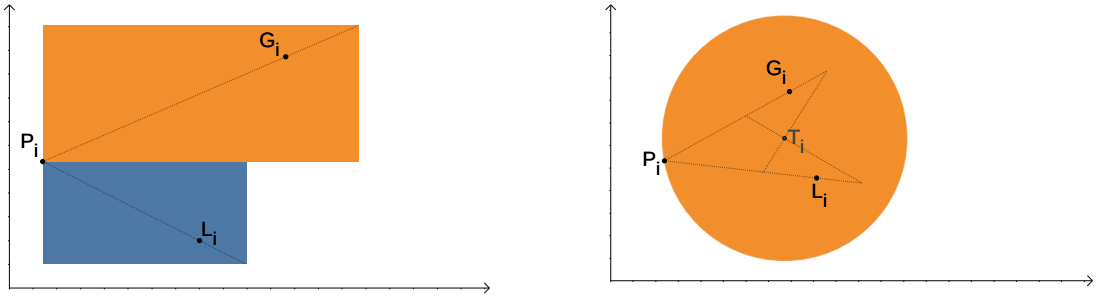
Some of the popular neighborhood topologies are ring (Figure 2.9a), dense ring (Figure 2.9b), and star (Figure 2.9c). Moreover, particles can be organized in the grid and communicate only with the surrounding particles. Some popular grid topologies are linear (Figure 2.10a), diamond (Figure 2.10b), and compact (Figure 2.10c) [**?**].

Notice that all these topologies are static and do not change during the run of the algorithm. Dynamic topologies were proposed, such as random topology and closest neighbors topology. In the random topology, each particle informs k random particles from the swarm. The k is usually kept small, for example $k = 3$ [**?**]. Note that different neighborhood is generated after each step. In the closest neighbors topology, each particle informs k closest particles. Except this topology is very computation–intensive, it usually leads to premature convergence because particles begin to create clusters and share their position only within it. On the other hand, it has prerequisites to find multiple local optima in a single run.



(a) Linear grid topology (b) Diamond grid topology (c) Compact grid topology

Figure 2.10: PSO grid topologies. Population is colored blue, active particle orange, and its neighborhood by green.



(a) SPSO–2006 sampling space (b) SPSO–2011 sampling space

Figure 2.11: Sampling space of SPSO–2006 and SPSO–2011

2.4.2 Velocity update

Difference between SPSO–2006 and SPSO–2011 is the way they update velocity. In SPSO–2006, the velocity update is

$$\mathbf{v_i} = \omega \mathbf{v_i} + c_l U_d\,(0,1) \cdot (\mathbf{L_i} - \mathbf{p_i}) + c_g U_d\,(0,1) \cdot (\mathbf{G_i} - \mathbf{p_i})$$

where ω is inertia weight, c_l and c_g are cognitive, respectively, social acceleration coefficients, and $U_d(0,1)$ is independent and uniformly distributed random d-dimensional vector in the range $[0,1]$. The $\cdot$ is an element–wise multiplication. The inertia weight prevents the velocity explosion, constants c_l and c_g controls how much the local best, respectively, the global best attracts the particle. These parameters are specified by the user. The particle position is then updated simply by applying new velocity on the particle.

$$\mathbf{p_i} = \mathbf{p_i} + \mathbf{v_i}$$

One may further limit the particle velocity or position, so it stays in the interval of interest. We may limit the velocity using its magnitude (then no particle can move faster than a specified threshold) or per coordinate, similarly to the position [**?**].

Because the velocity update in SPSO–2006 is done per dimension, the algorithm performance is rotation sensitive and coordinate system dependent. This phenomenon is visible in Figure 2.11a. The sampled point for the local and global

best position is sampled from the blue and orange region, respectively. As these regions align with axes, the resulting sampling is aligned with axes as well. It has been shown by **?** that it is easier for SPSO–2006 to find optimum if it lies in the center of coordinate systems or on the axis.

SPSO–2011 address this issue by sampling the point from hypersphere. First of all, it finds center of gravity $\mathbf{T}$ between current, local best, and global best positions.

$$\begin{aligned}\mathbf{l_i} &= \mathbf{p_i} + c_l U_d\,(0,1) \cdot (\mathbf{L_i} - \mathbf{p_i}) \\ \mathbf{g_i} &= \mathbf{p_i} + c_g U_d\,(0,1) \cdot (\mathbf{G_i} - \mathbf{p_i}) \\ \mathbf{T_i} &= \frac{\mathbf{p_i} + \mathbf{l_i} + \mathbf{g_i}}{3}\end{aligned}$$

A random point is then sampled from the hypersphere with the center $\mathbf{T_i}$ and radius $\|\mathbf{p_i} - \mathbf{T_i}\|$.

$$\mathbf{u_i} = \mathcal{H}_i\,(\mathbf{T_i}, \|\mathbf{p_i} - \mathbf{T_i}\|)$$

The following velocity update simply adds direction to the sampled point to the previous velocity.

$$\mathbf{v_i} = \omega \mathbf{v_i} + \mathbf{u_i} - \mathbf{p_i}$$

The sample space for two dimensions is shown in Figure 2.11b.

3. GPU Programming

Moore's law has driven the rise of processor performance for a long time — "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain constant for at least 10 years" [**?**]. This "law", if we may call it that way, is unfortunately limited by the physical properties of components used within modern processors. These boundaries are both on the side of transistors size and their quantity in the processor core, as well as on the side of maximum processing speed. Over the last years, scientists are predicting the end of Moore's law [**?**], and their voices are more turbulent since the size of transistors in the Central Processing Unit (CPU) reached $7nm$ in 2018 [**?**].

Processor manufacturers are well aware of the physical complications stemming from the transistors small size and invest more effort into multi–core processors. These processors have many independent cores that can be utilized in parallel, and can possibly increase the processor performance linearly with respect to the number of cores. To name a few, last Intel server processors Xeon® Platinum 8380 Processor with 40 cores and 80 threads [**?**], as well as AMD EPYC™ 7763 with 64 cores and 128 threads, are great examples [**?**].

Graphical Processing Units (GPUs) take this idea of multi–core processors further. Rather than increasing single–thread performance, the GPU manufacturers are focusing on massive parallelism using a vast number of cores operating in highly parallel and distributed manner [**?**]. Using this strategy, the GPU programming offers encouraging performance for heavy workloads. Common consumer GPUs can easily outperform current high–end CPUs in the instruction throughput and memory bandwidth. These demands came originally from the game industry, which places great emphasis on parallel processing and reasonable real–time latency.

The decisions regarding the architecture, though, affect the way GPUs are programmed. Unlike a CPU, which can run generic code, a GPU is more suited for specific classes of algorithms that can fully utilize the underlying hardware. Engineers recognized the following mandatory properties of the efficient application running on GPU [**?**]:

- *Large computation demands* — GPU can deliver tremendous performance but may be slow for short tasks without sufficient data. The overhead of memory allocation on GPU and moving it forth and back to CPU can quickly outweigh the advantage of using it. On contrary, the real–time rendering requires hundreds of operations per pixel, and GPU can easily meet these demands. GPU is furthermore hiding memory access latency behind more computations (I will talk about it a bit later); it therefore needs sufficient amount of work to do that.

- *Significant parallelism* — the algorithm needs to parallelize easily and ideally scale to as many cores as possible. Whereas CPU provides tens of cores (up to 64 today), the GPU has hundreds or thousands of them (the current cutting-edge GPU produced especially for AI, NVIDIA V100, has 5120

CUDA cores [?]), although with limited per–core performance. If an algorithm can employ only a few of them, it may be slower than on multi–core CPU. Once again, real–time rendering can render each pixel independently, making it a perfect candidate for GPU computing.

- *Throughput over latency* — because the GPU is highly parallel by design, the throughput is more important than the absolute latency of individual operations. From the point of rendering, the human eye can perceive images in order of milliseconds. Operations in modern processors take in the order of nanoseconds. Therefore, the absolute time to render a single pixel is not relevant as long as the whole picture is rendered below noticeable time. The algorithms cannot make assumptions about the running time of individual parts but rather on the task as a whole.

Evolutionary Algorithms fit well into these categories, as the algorithm may evaluate individuals' fitness in parallel, and the evaluation is independent with respect to the rest of the population. The crossover and mutation operators may be easily parallelized as well, because they operate on individuals (or a small set of individuals) independently. We may parallelize the algorithm even more by processing each gene separately, as presented by **?**. I will discuss this option at the end of this chapter.

3.1 History

Historically, GPU came from the demand of the game industry for real–time rendering. In the rendering process, a list of geometric primitives, usually triangles, is processed through a number of stages to produce a final picture. The primitives can be processed independently and therefore in parallel, which makes real–time rendering the ideal case for the application of GPU. The typical operations in graphical pipeline consists of [**?**]:

- Vertex stage that transforms and calculates properties per vertex. Typical operations are transforming vertex position from world space (that is, the coordinates in the scene) into screen space (that is, the coordinates with respect to the viewer point of view) and estimation of additional vertex properties like color, material, or texture coordinates.

- Primitive assembly transforms geometric primitives into triangles – the standard geometric primitive the GPU can work with. This stage may also clip the primitives that are not in the observer's view and save some computation in the following stages.

- Rasterization takes generated triangles and resolves which pixels correspond to the provided triangle. These pixels do not necessarily need to match the number of pixels of the screen. They are usually referred to as fragments. For example the superscaling technique renders the scene in higher resolution and then downsamples it to match screen resolution [**?**]. Each triangle may be composed of hundreds or thousands of fragments. Moreover, the rasterization typically interpolates vertex properties between the fragments.

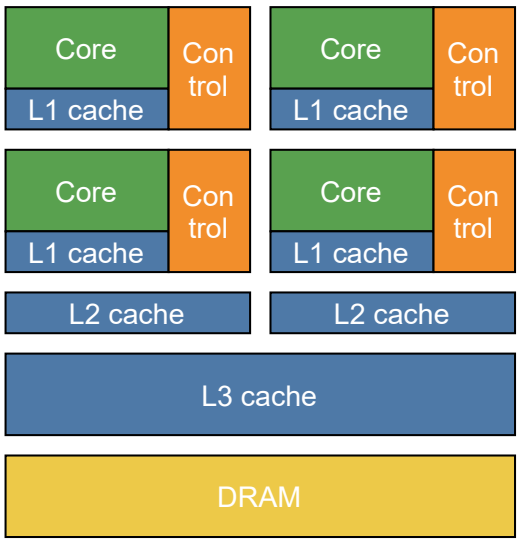
- Fragment stage processes each fragment and typically resolves the final appearance of the fragment. It may calculate fragment interactions with the lights in the scene, fetch colors from textures, blend fragments, etc. This stage is usually the most demanding, as a typical scene consists of millions of fragments.

- Composition stage assembles all the fragments into a final image. It may, among others, test fragments visibility, depth, and do stencil filtering.

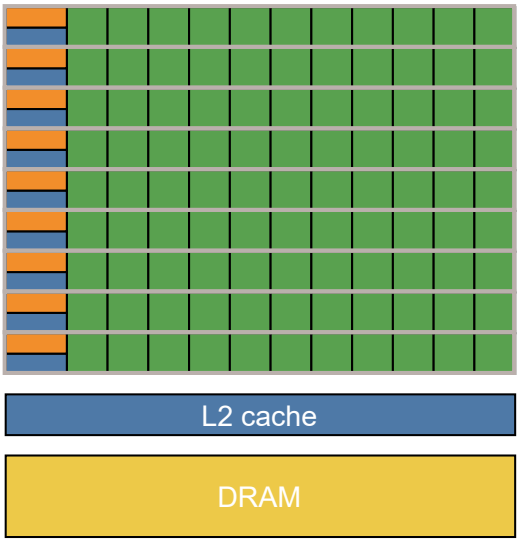
The stages GPU performs are usually referred to as *rendering pipeline* and may have slight variations on different hardware. At the beginning, the stages discussed earlier have been preprogrammed by the manufacturer and developers had only limited options how to control the pipeline stages (also known as the fixed–function pipeline). As the software complexity increased, the need for more control arised, and the manufactures provided a way to program the individual stages of the pipeline. These programs are called *shaders* and their capabilities matured over the years. In 2006, Microsoft introduced Shading Model 4.0, which allowed using the same programming interface and hardware both for vertex and fragment shaders [**?**]. Moreover, more stages appeared (like tessellation and geometry shaders) to provide developers more control over the pipeline. Today, two major API exists — proprietary DirectX and open–source OpenGL. Note that some stages of the rendering pipeline, such as primitive assembly, rasterization, and composition, are still implemented by the GPU manufacturers and, in most cases, in special–purpose hardware components (because of the performance reasons) [**?**]. Hence, they are the integral components of the rendering pipeline and are not possible to modify.

Soon, scientists and engineers recognized many more algorithms that would benefit by GPU performance. Although some stages of the rendering pipeline were fully programmable, the pipeline was still inherently graphical, and some stages were not possible to skip. Executing general–purpose programs on GPU required encoding the problem in the domain of geometric primitives and use of available programmable and special–purpose hardware to implement the algorithm. Moreover, the programmers need to decide which part of the hardware will execute which part of the calculation and how the fixed stages of the pipeline will affect the data structure. As shader programs do not have access to shared memory, the data passing was realized using textures, vertices, and fragment properties [**?**]. These difficulties discouraged wider adaption of GPU for general–purpose programs, although some efforts have been made to hide the underlying rendering pipeline from the programmer point of view [**?**].

The lack of support for general–purpose program on GPU inspired companies to develop a new approach that would provide programmers a better control over the underlying hardware. In 2006, NVIDIA introduced the first version of the Compute Unified Device Architecture (CUDA) [**?**]. Open–source standard came in 2008 as Open Computing Language (OpenCL) by Khronos Group [**?**]. These are higher–level interfaces with C–like syntax exposing GPU resources without any connection to the graphical pipeline. Programmer is given control of execution and parallelization of the program, as well as memory access. Over the years, these technologies have undergone various improvements (the current version is CUDA 11.2, OpenCL 3.0, respectively), adding support for data types,



(a) Architecture of CPU. Each core has L1 cache and own control circuit. Cores can therefore work independently.



(b) Architecture of GPU. One control circuit controls number of cores. One row of cores is called Stream Multiprocessor (SM, bounded by gray borders). L2 cache is shared between the cores.

Figure 3.1: Difference between CPU and GPU architecture

language constructs, and new hardware.

CUDA become de facto standard in the field of general–purpose computing on GPU and I will focus mainly on it. Fortunately, other technologies like OpenCL have similar underlying architecture and approach to the GPU programming. Hence, the following section should generalize for them as well.

3.2 Architecture

The design goal of GPU is to provide orders of magnitude higher instruction throughput and memory bandwidth compared to CPU. The overall architecture of GPU is therefore significantly different. This section will focus on GPU architecture from the point of CUDA platform, taking inspiration mainly from the CUDA Programming Guide [**?**].

The computation power of GPU emanates from the fact it dedicates more transistors on the chip to the data processing rather than to the control circuitry. The architecture of the CPU is shown in Figure 3.1a. Each core has its own control circuit, and therefore each core can execute instructions independently. Moreover, each thread has its own L1 cache.

The GPU architecture is portrayed in Figure 3.1b. A set of cores share the same control circuitry and L1 cache. One line of these cores is called Stream Multiprocessor (SM). The exact number of cores per SM differ between architectures; more information can be found in the article by **?** or on NVIDIA official websites. Using one control circuitry to manage many cores brings some implications – all the cores need to execute the same instruction at once. In this sense, the instruction throughput is achieved using Single Instruction, Multiple Data (SIMD) approach – one instruction can at one tick alter multiple data on multiple cores.

Each GPU comes with a different set of hardware and software features. Compute capability specifies which features are available on the target system. For example, half–precision float–point operations are available since compute capability 5.3. The number of arithmetic CUDA cores per SM is 8 for compute capability 1.x, 32 resp. 48 for compute capability 2.0 resp. 2.1, 64 for compute capability 6.0, 7.x, and 8.0, 128 for compute capability 6.1, 6.2, and 8.6, and finally 192 for compute capability 3.x. I will not discuss specific compute capability in the following text. I will assume compute capability of at least 6.0 through this work, but I do not depend on the concrete version, as they should be backward compatible. In order to use newer compute capabilities, the GPU, driver, and the software need to support them.

Compute Unified Device Architecture allows almost direct translation of code written in C++ onto the GPU. It uses C++ language with a few extensions. It is therefore straightforward to write a program for GPU, as long as we are familiar with C syntax. To distinguish code running on CPU and GPU, CUDA differentiate between the host (the code running on CPU) and device (the code running on the GPU) functions. There is a particular type of function called *kernels*, which is evoked by the host but executed on the device. The kernels are the main components to execute code on the GPU.

The CUDA defines the following terminology:

- CUDA thread is the execution of a single kernel. To process K blocks of data, the runtime creates at least K threads and runs the same kernel function on each of them.

- CUDA thread block is a collection of threads grouped into the grid. The programmer may specify the size of the grid during the call of the kernel function as a three-dimensional vector. The maximum number of threads in the thread block is limited by the compute capability. Moreover, all the threads in the thread block need to be executed on one SM (with some exceptions) and can be synchronized using *__syncthreads* call. Each kernel receives *threadIdx* variable, which allows the identification of individual threads in the block.

- CUDA kernel grid is a grid of thread blocks. Similarly to thread block, its size may be specified during kernel invocation, and each thread receives *blockIdx* and *blockDim* variables identifying block in which the thread is running and size of the blocks, respectively. More blocks from the same grid may run in parallel on different or same SM (if the SM have free capacity), but the threads from distinct blocks cannot be synchronized.

- Warp is a set of 32 threads (for all compute capabilities so far) that execute the same instruction. The thread block is split into warps and executed by the SM. The warp is the minimal unit executable on the GPU . The device plans warps the same way OS plans processes and may switch context to another warp while the current one is waiting for the data. By executing different warp, the latency of, for example, memory access is "hidden" behind the execution of different warp – known as hiding latency behind computation. Although GPU cannot execute anything smaller than the warp (like a single core), it may mask some threads from the execution. Therefore, it is

possible to have a block size of 50 threads; however, $2 \cdot 32 - 50 = 14$ cores in the second warp will still execute the kernel without any effect (this matter is hidden from the programmer), becase they will be masked out.

Because CUDA assumes a system composed of host and device, each of them has its own separate memory. The kernel executes on the device and, as such, needs to have data its operating on moved into the device memory. CUDA provides a set of functions to allocate, deallocate, copy, and move memory between the host and the device. Similar to traditional C++ programming, CUDA memory is allocated in a single unified linear address space. It can reference other addresses using pointers and use advanced data structures like linked lists and trees. Yet, the data movement between host and device is time–consuming, because the bandwidth between CPU and GPU is lower than memory transfers within the device memory. The memory copy from the host to the device may generate nontrivial overhead that may waive the advantage of using GPU altogether, especially for small kernels.

To reduce access latency to global device memory, CUDA provides a set of tools to speed up the process. It is possible to control the L2 cache policy to better match workloads of executing kernels. It also provides functions to manage *page–locked host memory*, which remains in the host memory and will not page by the OS. Copying from this kind of memory can be done asynchronously, or when allocated as *mapped–memory*, it is shared between the host and the device. Mapped–memory is implicitly transferred back and forth the device, and the programmer does not need to allocate it explicitly.

A special kind of memory, residing on–chip, is *shared memory*. The latency of shared memory is roughly 100x times lower than for global device memory, and the memory is shared in the thread block. Simple operations like matrix multiplication can hasten by orders of magnitude using shared memory [**?**]. However, shared memory operates in equally–sized memory modules called banks, which complicate concurrent access to the shared memory. All the compute capabilities up today have 32 banks for shared memory organized into 32–bit words. The access to distinct memory addresses belonging to the same bank (bank conflict) by multiple threads within the same warp is serialized and, therefore, significantly increases the kerne running time. This is common for data types with 64–bits, such as double–precision floats. It may be beneficial to pad each value with one byte to eliminate conflicting access to the same banks.

As I mentioned earlier, all threads in the warp need to execute the exact same instruction. However, the programming language supports conditional jumps, loops, and other constructs known from the modern programming languages. It may certainly happen that some threads within the warp may execute different execution path – a situation known as *warp divergence*. There are further causes, why the threads in the warp may diverge, but branching is the most common one. In order for GPU to evaluate each execution path, it executes the branched paths sequentially and masks out threads not participating in the current one. Masked threads will execute the program; however, their results are ignored from the programmer point of view. This implies the same code will be executed twice (note that this grows exponentially for nested conditions) and significantly extend the time to finish the program. In general, warp divergence is undesirable and should be avoided when possible. Warp divergence can only occur within a warp,

different warps in the same thread block may execute different execution paths without any impact on the program performance.

The parallel nature of CUDA helps to speed up several algorithms that have by order of magnitudes better performance on GPU than on CPU. A typical example would be element–wise linear operations or matrix multiplication [**?**]. Reduction operations such as array summation can be implemented effectively on GPU [**?**] as well as different kinds of sorting algorithms [**?**].

These operations are so popular, NVIDIA released a set of libraries with their implementation. The most relevant for this work are *cuBLAS* (GPU–accelerated basic linear algebra library), CUDA Math Library implementing most common mathematical functions, and *cuRAND* (CPU–accelerated random number generation). These implementations are optimized for the latest hardware and maintained by NVIDIA.

3.3 PyTorch

From the brief description of CUDA above is clear that programming in CUDA is fundamentally different from programming in traditional programming languages like C or C++. The efficient implementation of kernels is complex, and in order to fully use the power of GPU, the programmer needs to understand the underlying hardware. Not an ability commonly found among scientists and engineers dealing with evolutionary algorithms. As one of the goals of this thesis is to provide an easy-to-use and extend library for evolutionary algorithms, I decided to look for an alternative.

PyTorch is free open–source library maintained by Facebook AI Research lab (FAIR). It builds on top of the Torch library for scientific computing on GPU. It is possible to use C++ or Python programming languages to implement applications using PyTorch library, where the Python language is the primary language the library focuses on. I argue more programmers are familiar with Python language than with C or C++; and its use is hence more plausible than CUDA for the purpose of this thesis [**?**]. PyTorch hides the underlying kernels behind a unified interface, and the programmer does not need to know CUDA programming at all. When necessary, PyTorch allows implementing parts of the algorithm in C++ or even as native CUDA kernels, and chain it with the rest of the PyTorch library [**?**]. The end–user may call this native implementation directly from Python source code without even noticing it. This approach allows optimizing specific parts of the algorithm if the performance is not pleasing.

The building block of PyTorch library is the `torch.Tensor` data type. It represents a multidimensional array of scalars. PyTorch allows to define its type to be the same as supported by most of the programming languages. The most common are 16, 32, or 64–bit floating–points numbers (also known as half–precision, single–precision, and double–precision floating–point formats), signed 8, 16, 32, and 64–bit integer numbers (also known as char, short, integer and long numbers), 8–bit unsigned integer (known as byte), and Boolean data type storing either 1 (true) or 0 (false) value. Except for these, PyTorch supports 32, 64, and 128–bit complex numbers.

Tensors are efficiently stored in memory by setting the size and stride of each dimension. This allows representing k-dimensional tensor of a constant value by

a single, in memory value. We set the desired size of each dimension and stride equal to 0. Because PyTorch kernels anticipate tensors in this format (with size and stride defined), they will automatically reference the single value instead of duplicating it in the memory. There is also the possibility to store sparse tensors using the `torch.sparse` package and keep in memory non–zero values only. Note that sparse tensors are only in beta version [**?**].

The kernels are realized as operations with tensors. PyTorch provides a broad range of operations, highly similar to the functions in the NumPy library, which is de facto standard for numeric computation algorithms in Python. These functions are also similar to MATLAB numerical functions, and very presumably, programmers coming from other platforms may effectively implement their algorithms in PyTorch because of the familiar API. This is one of the reasons why I chose PyTorch as the library for the implementation of evolutionary algorithms.

PyTorch functions realize standard algebraic operations, geometric and statistical functions, matrix multiplication, bitwise operations, reduction operations like summation, finding unique elements, or logical testing. Moreover, PyTorch implements functions from BLAS (Basic Linear Algebra Subprograms) specification and LAPACK (Linear Algebra Package) library like matrix decomposition, linear equations solver, determinant, and much more [**?**].

For pseudo–random number generation PyTorch uses underlying implementations — either Mersenne Twister for CPU or Philox implemented in cuRAND library. The latter is efficient for CUDA programming, but needs to be initialized by each thread before the generation. Historically, people were generating random numbers on CPU and moving them to GPU afterwards. Using native CUDA implementation is without a doubt more effective.

The advantage of PyTorch is that its functions are implemented both for CPU and GPU. It is possible to use the same source code without any changes on machines without GPU. The implementation may still use SIMD instructions on CPU, that are ordinarily available on modern processors (also known as vectorized instructions). Note that although these are still SIMD instructions, the performance is not nearly as good as running all the computations in parallel on the GPU. Nevertheless, the performance would still be a bit better than using C or C++ purely without optimization.

Another advantage of using an existing library like PyTorch is that the implementation (and especially the CUDA kernels) is written and maintained by a team of professionals that understand the architecture and working of the GPU most likely better than most of the people dealing with evolutionary algorithms. Using existing frameworks gives assurance the kernels are implemented efficiently both on the GPU and CPU. What is, in my opinion, more important is that the library is kept up-to-date and may use features of new, not yet released hardware, in the future. All we need to do is to update the library, and the same source code may use the new hardware and features.

Because of the PyTorch properties mentioned above, I decided to implement the evolutionary algorithms in this library.

3.4 Evolutionary Algorithms Parallelization

General Evolutionary Algorithms manage a population of individuals and by repeated application of evolutionary operators search for the optimal solution of the problem at hand. As the individuals within the population are independent, there is an excellent potential to speed up the whole process by parallelizing evolutionary operators. Although some of the operators need more than one individual for their execution (the crossover operators are the typical example), the operator is, in most cases, applied several times on different sets of individuals. Further, these sets are small, and the operator may be executed in parallel for all of them. The typical crossover operator, for example, needs only two individuals and is applied repeatedly on different parents.

Scientists and researchers were well aware of this potential, and the first attempts [**?**] to parallelize evolutionary algorithms came long before CUDA was introduced by NVIDIA in 2006. These projects were either using a single machine with multiple processor cores, or a system of machines sharing the population over the network. The common idea is to divide the population into chunks and solve them simultaneously using multiple processors. There are two ways of doing that [**?**]:

- Master–worker model executes all the evolutionary operators on a single machine, but the fitness evaluation is spread among several processors. While the fitness evaluation takes a significant amount of time (for example, run of a physical simulation is very time–consuming), and the time to share the individuals among processors is negligible, the master–worker model is the best way to accelerate the algorithm.

- Island model maintains a list of populations evolving independently. Each of these populations may run on a separate processor and therefore in parallel. The populations are usually called *islands* because from the evolutionary point of view they appear as evolutions on different continents. The islands do not collaborate with each other, except a specific operator called *migration operator*. This operator periodically, in a predefined way, exchanges a small portion of the population among the islands. By borrowing few individuals from a different island, we hope to bring new genetic material. It has been shown that the island model on its own may improve the solution quality and running time of the algorithm [**?**].

These models are historically more suited for multi–processor systems rather than for the GPU, as it would be hard to scale them up to thousands of cores currently available in modern GPU. Nevertheless, these approaches can still be combined with the proposed implementation. One may, for example, use the island model and run each population on a different GPU–enabled machine.

In parallel computing, the granularity is a measure of the computation amount conducted by the task. Typically, there is coarse–grained parallelism, which splits the problem into several large tasks. Fine–grained parallelism, in contrast, breaks the problem into a large number of small tasks. Both coarse–grained and fine–grained algorithms have been proposed for CPU architecture. For GPU programming, however, more delicate level of granularity is needed:

- Chromosome–level granularity evaluates one individual using one thread. Master–worker model mentioned earlier is an example of the chromosome–level granularity. For CUDA purposes, this degree of granularity is still too raw.
- Warp–level granularity evaluates one individual using single warp. The advantage of this case is the possibility for threads to communicate, and for some problems, that may be necessary to do so.
- Gene–level granularity evaluates each gene using one thread, or more generally evaluates one individual using more threads (then warp–level granularity is just a special case of the gene–level granularity).

The gene–level granularity is ideal for the GPU computing, as it exposes enough parallelism. Earlier attempts with chromosome–level or more coarse level granularity could not fully utilize the GPU. One example of the gene–level granularity is in the work of **?** for the knapsack problem. His design of the CUDA kernel is in Figure 3.2. Other authors implemented various algorithms using CUDA, for example, simple genetic algorithm [**?**], differential evolution [**?**], and particle swarm optimization algorithm [**?**]. All of them showed orders of magnitude speedup and proof that implementing Evolutionary Algorithms on GPU has its advantages.

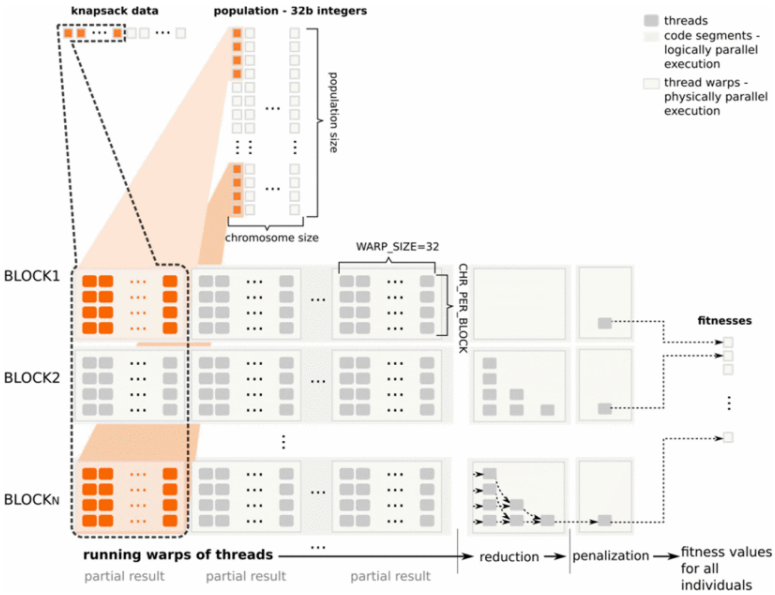


Figure 3.2: The knapsack fitness function kernel in the work of **?**. It is gene–level model, where each thread aggregate informations from one individual and CHR_PER_BLOCK items.

4. Proposed Implementation

My goal is to design and implement a library for Evolutionary Algorithms, as mentioned in Chapter 1. The library should not only implement existing EA and allow to execute them on the GPU, but also afford a general framework to which it would be simple to plug new algorithms. For library users should be easy to replace evolutionary operators, modify them, or, if necessary, replace the whole workflow of the algorithm.

I decide to implement the library using Pipe and Filter architecture as specified by authors **?**. The example of this architecture is in Figure 4.1. The general idea is to partition the algorithm into smaller, simple steps and use the output of one step as the input to the following one. EA are simple to split as one step may be one evolutionary operator. I decided to implement the library in the "Convention over Configuration" design pattern – that is, the operators make the same assumptions about the format and order of the input, rather than explicitly specifying it in the configuration. I still provide ways to control the order of parameters, if required by the user, and I will get to it a bit later.

The pure Pipe and Filter architecture is not sufficient for EA implementation. In the purest implementation, this architecture is just a chain of steps following each other. I decided to expand it by numerous practical steps like loops, parallel step invocation, and more. An example of parallel step and loop used in the Pipe and Filter architecture is demonstrated in Figure 4.1. The parallel invocation is depicted in the figure by "Parallel" block, followed by the "Aggregation" block accumulating the results from each step. The loop is represented by the "Repeat" block accompanied by the "Termination" block controlling the loop termination condition.

My proposed implementation is the *Framework For Evolution Algorithms in Torch* (FFEAT). This library is attached to this work, available on my GitHub [**?**], and ready for install as PyPI (Python Package Index) package. The PyPI is de facto standard way of installing Python packages from a central repository.

As I mentioned earlier, I decided to implement the library in the Python programming language and using PyTorch library. Python allows invocation of function with a variable number of arguments and a variable number of keyword arguments. Without explaining it in too much detail, arguments are passed into the function in a list and depend on their order. These are the traditional arguments known from languages like C and C++. Keyword arguments must be specified by their parameter name. The following line of code invokes function

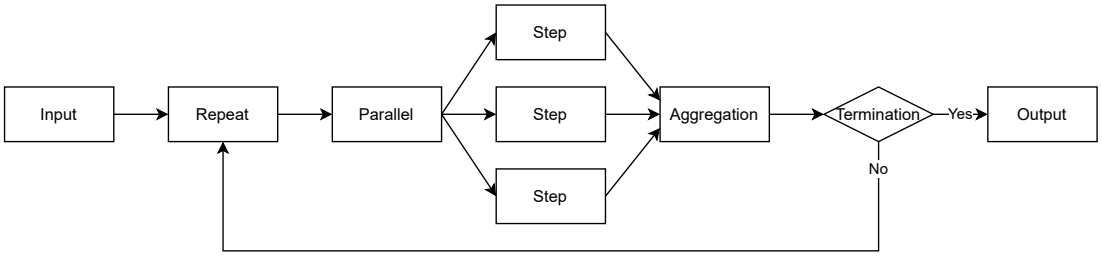


Figure 4.1: Pipe and Filter architecture

some_function with two parameters 1 and 2, and a keyword argument $karg = 5$.

```
some_function(1, 2, karg=5)
```

I use this design to implement operators in Pipe and Filter architecture. The operator accepts a variable number of normal and keyword arguments, and returns a list of variables and a dictionary. The list of variables and the dictionary serves as the input for the following operator as normal, respectively, as keyword arguments. Following this design, a new operator may be easily plugin into the algorithm. At the same time, it allows using the same underlying architecture both for genetic algorithms (the argument is only the population), as well as PSO algorithms (using particles position, velocity, and their best–known positions as arguments).

The library expects the population to be represented as a tensor or as a list of tensors, where the first dimension enumerates over the individuals (note that the tensor needs to be aligned in each dimension). I resolve to this assumption because it allows me to implement the operators in vectorized manner and, therefore, more efficiently. Nevertheless, the architecture is general enough, and it may easily adjust to the problem at hand; the user would just need to reimplement the operators when necessary. In most cases, it is enough to represent the population using a single, two-dimensional tensor. Some operators are implemented in a way they may handle multidimensional tensors as well (for example the uniform crossover). In the case of PSO algorithm, using a single tensor is not sufficient. The library represents the swarm using a list of tensors – one tensor for each component of the population. Specifically, the library creates tensors for particles position and velocity, local and best–known positions, and their corresponding fitness values. Although the PSO uses list of tensors, each of them still fulfills the requirement to use the first dimension to enumerate over individual particles. The operators can hence effectively run in parallel.

The library is split into several modules, focusing on different kinds of evolutionary algorithms or solving particular problems.

- `ffeat.flow` module includes basic classes to control the flow of the algorithm with the Pipe and Filter architecture in mind.
- `ffeat.genetic` implements genetic algorithms operators and assume binary encoding.
- `ffeat.strategies` contains all the real–coded operators. Although not all of them are adaptive, therefore, they cannot be considered as evolution strategies, I decided to put them into the shared module for convenience.
- `ffeat.pso` handles Particle Swarm Optimization algorithms, their neighborhood and velocity update algorithms.
- `ffeat.measure` implements aggregations functions focusing on fitness metrics.
- `ffeat.utils` is a support module containing various useful functions like fitness scaling, decay, and early termination implementation.

The base class for all the operators is `ffeat.Pipe`. The implementation merely accepts all the arguments and returns them. The specific logic needs to be provided by the derived class. The method the library invokes is the Python `__call__` method allowing to call the object in the same way as if it was a function. That means the library invokes the operators using the standard invocation procedure, allowing the implementation of simple operators to be a function or a lambda function rather than a class.

The `ffeat.flow` implements basal classes to use the Pipe and Filter architecture. I will refer to operators to invoke as "filters" to agree with the terminology used in the architecture. The most important filters are:

- `Sequence` executing filters in sequence and passing the output of the preceeding one into the following. The `Sequence` class alone implements the simplest Pipe and Filter architecture.
- `Parallel` executes filters in parallel – that means all of them receive the same parameters, and their results are concatenated together. The execution is parallel from the logical point of view, not using multi threading.
- `Repeat` executes filters in a loop for a given number of iterations (or indefinitely). The implementation allows breaking the loop early by passing "break" parameter as a keyword argument into the inner filters.

The module encompasses few more classes allowing to reorder and discard parameters (`Select` class), replace some of them (`Replace` class), transform each parameter (`EachArg` class), and use Python lambda function (`Lambda` class). I do not discuss them here, as the details are not important. Please see the attached source code for more information.

Before I describe particular operators, I would like to discuss the limitations of the PyTorch library and GPU programming in general. As I already mentioned in Chapter 3, the most costly situation is warp divergence. I tried to implement all the operators without condition statements and, therefore, eliminate the warp divergence situation as much as possible. I found boolean mask as a suitable replacement for condition statements. PyTorch may use boolean mask either for indexing (if the only portion of tensor should be updated) or in an arithmetic operation by multiplying tensor by the mask (and zeroing portion of the tensor).

A further limitation is the GPU memory, which is a scarcer resource in comparison to the modern server machines. Modern graphical cards have up to low dozens of GB of memory (the latest NVIDIA V100 has 32GB of device memory [**?**]), whereas modern servers have hundreds of GB of memory. Because the library may use vast populations, even memory reallocation may be problematic, as the whole population must fit into the memory at least twice. I decided to implement the operators in–place, that is, all the operators are applied on the same memory. Not only it uses less memory, but it is also more efficient. Some operators (typically crossovers) have possibility to opt–out of in–place mode (for example, in the case of plus or comma crossover schema).

Operators using parents, for example crossovers and differential evolution, should receive distinct parents in order for the operator to do something useful. Unfortunately, this requirement is hard to satisfy, especially for GPU, because the parent indices are generated independently. It is, nevertheless, still possible

using the `torch.multinomial` function from the PyTorch library. The problem is that this function is very costly and its execution takes an order of magnitude more time than generating a random integer. I decided to allow specification of the parent sampling strategy for all these algorithms.

4.1 Genetic Algorithms

The Genetic Algorithms are in the `ffeat.genetic` module. This module is further divided into the following submodules.

- `ffeat.genetic.initialization` holds classes initializing the population. I considered only random initialization and this is implemented by the `Uniform` class.
- `ffeat.genetic.evaluation` submodule contains classes to evaluate the individuals. The `Evaluation` class expects to evaluate the whole population at once (ideal for GPU implementation), whereas the `RowEval` evaluates individuals one by one. Both classes expect to receive fitness function in their constructor.
- `ffeat.genetic.mutation` includes mutation operators. The library only implements simple Bit–Flip mutation operator, allowing to specify a number of mutated individuals and the probability of gene change.
- `ffeat.genetic.crossover` submodule.
- `ffeat.genetic.selection` submodule.

I implemented uniform, one–point, and two–point crossovers for GA (classes `Uniform`, `OnePoint1D`, and `TwoPoint1D`). The uniform crossover may handle individuals with an arbitrary number of dimensions because it simply generates a random mask of genes to inherit from the first parent. In the case of one– and two–point crossovers the crossover point would be ambiguous; these, therefore, expect individuals to be one-dimensional. The one– and two–point crossover implementation uses similar mask, but it is created based on the split point. That is a bit costly, especially for the two–point crossover, because each point must be added separately. The example implementation of one–point crossover is in Algorithm A.2.

All the crossover operators implement three distinct schemes of how to handle the offspring. The default scheme is the in–place version and the offspring replace their parents. For this to work, the crossover must produce the same number of offspring as there are parents. Moreover, the crossover operator should not process the whole population, as it may destroy it or shift the whole population into an undesirable region of the search space. Still, I decide to use it as the default version because of the GPU limitations mentioned above. The second and third schemes are the comma and plus schemes mentioned on Page 13. The plus scheme concatenates the offspring with the parents and is specified by the `replace_parents=False` argument. The comma scheme returns just the offspring and is specified by the `discard_parents=True` argument passed to the crossover constructor.

In the `ffeat.genetic.selection` module the library keeps implementations of the tournament (`Tournament` class) roulette (`Roulette` class), and Stochastic Universal Sampling (`StochasticUniversalSampling` class) selection operators. The tournament selection allows to specify whether is it maximization or minimization problem and a number of parents as I discussed in Chapter 2. Its implementation is in Algorithm A.1. The rank–based selection operator is in fact the roulette or SUS selection with fitness values preprocessing. I will get to fitness transformation later.

I also implemented elitism in the `ffeat.genetic.selection.Elitism` class. It copies the n best individuals from the population and temporarily stores them aside. After all the GA operators execute, it copies them back into the population. The implementation does not follow the exact description found in books [**?**], because it may replace better individuals (when the elite improves). The standard implementations of elitism prohibit elites change by the following operators, or append the elites to the population after all the operators executed. Because the operators following the elitism may be arbitrarily complicated, or the user of the library may decide to implement their own set of operators, ensuring the elites would not change would be thorough and delicate work. Moreover, from the GPU programming point of view, copying a small amount of memory between an already allocated memory space is more efficient than appending the elites to the population, because that would require reallocation of the whole population. The implementation of the elitism operator is in Algorithm A.3.

For convenience, the parameters dealing with population size, typically the

Algorithm 4.1: Simple GA in FFEAT

```
import ffeat.genetic as GA


fn = create_problem_function()

alg = GA.GeneticAlgorithm(
    # Randomly initialize 100 individuals
    # with gene of length 40
    GA.initialization.Uniform(100, 40),
    # Evaluate the population
    GA.evaluation.Evaluation(fn),
    # Sample 100 individuals into new generation
    GA.selection.Tournament(100),
    # Crossover 40% of them
    GA.crossover.OnePoint1D(0.4),
    # Mutate 60 of them with 1% mutation chance
    GA.mutation.FlipBit(60, mutate_prob=0.01),
    # repeat for 100 generations
    iterations=100
)
alg() # run the evolution
```

number of individuals to sample during selection, or the number of offsprings in crossovers, can be specified using an absolute number or a fraction of the population size. Moreover, some operators parameters may change between the generations, for example mutation probability of Bit–Flip mutation. These parameters accept callable object evaluated each iteration, allowing them to adjust. Finally, there is no need to use `ffeat.flow` module directly, but the flow is wrap in the `ffeat.genetic.GeneticAlgorithm` class, into which the user just needs to plug the operators.

Example of a genetic algorithm in the FFEAT library is in the Algorithm 4.1.

4.2 Real–Coded Evolutionary Algorithms

Real–coded evolutionary algorithms are encapsulated in the `ffeat.strategies` module. It has a similar structure to GA module described above. Except for the operators mentioned above, I implemented operators specific to real–value encoding.

In the `ffeat.strategies.crossover` submodule are uniform, one–point, and two–point crossovers identical to the ones for GA. Besides, it contains:

- Arithmetic crossover in the `Arithmetic` class. It allows specifying the number of parents and their weights. For k parents, the offspring is created by the formula

  $$\mathbf{o}_i = \sum_{j=1}^{k} w_{ji} \mathbf{p}_{\mathbf{j}_i}$$

  It allows passing a callable object for the weights so that the weights may be randomized and the offspring may be different weighted arithmetic sum each generation (allowing different weights for genes as well).

- Blend crossover in the `Blend` class, as described in the Chapter 2.

- Differential evolution implemented in the `Differential` class. Although this operator may be used alone, I decided to keep it among the crossover operators. It is possible to alter the F and C constants over generations and replace the parent only if the offspring is better than its parent. In this case, the operator needs to know the fitness values of the parents.

From the mutation operators, I implemented the following operators in the `ffeat.strategies.mutation` submodule.

- Random replacement of gene by a value sampled from the specified distribution. It supports all the distributions implemented by the PyTorch library [**?**] and may refine the mutation rate over generations. It is implemented in the `Replace` class, along with the `ReplaceUniform` class sampling from uniform distribution.

- Small change of the individual by adding a value sampled from the specified distribution. It is encapsulated in the `AddFromDistribution` class with specialized classes `AddFromNormal` and `AddFromCauchy` for normal and Cauchy distribution, respectively. The `AddFromNormal` is the traditional implementation of normal mutation specified in the second chapter.

- Normal mutation with adaptive step implemented in the `AdaptiveStep` class. It supports only the simplest adaptive algorithm by sharing the same deviation for all the dimensions. It allows specifying initial, maximal, and minimal deviation, as well as the step size and a number of better offspring needed to increase the deviation. This class may be used to implement the one–fifth rule.

The `ffeat.strategy.EvolutionStrategy` class wraps together all the steps for real–coded evolutionary algorithms. Simple real–coded algorithm is in Algorithm 4.2.

4.3 Utility functions

I implemented some functionality allowing to control and measure the algorithm progress. The first set of these functions are in the `ffeat.measure` module. It allows measuring statistical data like mean, deviation, and quantiles of the fitness. It passes measured metrics as keyword arguments to the following operator, so it is possible to stop the algorithm early if a specific metric does not improve or a certain threshold has been reached. Also, it is possible to log these metrics into standard output or file, if necessary.

The responsibility of the `ffeat.utils.termination` submodule is the early termination of the algorithm. There are various termination criteria, for example

Algorithm 4.2: Simple real–coded algorithm in FFEAT

```
import ffeat.strategies as ES


fn = create_problem_function()

alg = ES.EvolutionStrategy(
    # Randomly initialize 100 individuals
    # in range (-5,5) with gene of length 40
    ES.initialization.Uniform(100, -5.0, 5.0, 40),
    # Evaluate the population
    ES.evaluation.Evaluation(fn),
    # Sample 100 individuals into new generation
    ES.selection.Roulette(100),
    # Crossover 40% of them
    ES.crossover.TwoPoint1D(0.4),
    # Use normal mutation for all of them with
    # standard deviation 0.01
    ES.mutation.AddFromNormal(0.01),
    # repeat for 200 generations
    iterations=200
)
alg() # run the evolution
```

if metric does not improve for a specified number of generations (`NoImprovement` class), metric deviation for last k steps is bellow a certain threshold (`StdBellow` class), or metric reached the specified threshold (`MetricReached` class).

The decay rates presented in Section 2.3.3 are implemented in submodule `ffeat.utils.decay` – specifically the module implements linear, polynomial, and exponential decay rates. This module allows, for example, to decrease the mutation rate over generations.

The last submodule is `ffeat.utils.scaling` which allows to rescale the fitness for the purposes of proportional–based selection operators. The fitness may be scaled linearly, exponentially, or using a logarithmic scale. The transformation of the minimization problem into the maximization one can be done using the `MultiplicativeInverse` class that changes fitness using the $f(x) = 1/x$ formula.

Earlier in this work, I have mentioned that rank–based selection operator is just a roulette-based selection operator with modified fitness values. The class `RankScale` conducts that by sorting the population by the old fitness values and giving each individual a new fitness value based on its order. The given fitness values are linearly distributed among the population, but may be transformed into a different scale using one of the above–mentioned classes.

4.4 Particle Swarm Optimization

The Particle Swarm Optimization algorithms are implemented in the `ffeat.pso` module. It contains the implementation of both SPSO–2006 and SPSO–2011 algorithms, along the following neighborhoods in the `ffeat.pso.neighborhood` submodule.

- Random neighborhood.
- Nearest neighbors neighborhood.
- Static neighborhood is a helper class that caches the neighborhood in the first generation and uses it for the rest of the run. The following neighborhoods use the static neighborhood as their base class, and therefore the time to build the neighborhood does not affect the algorithm running time.
- Circle neighborhood with the possibility to specify the number of neighbors.
- Grid neighborhood with either linear, compact, or diamond shape, as specified in Chapter 2.4.1. The compact and diamond shapes are tricky to build in an arbitrary number of dimensions, so these are implemented only for two-dimensional cases and encapsulated in the `Grid2D` class.

In the `ffeat.pso.clip` submodule are classes allowing to clip either particles position or velocity. The whole PSO algorithm is wraped inside the `PSO` class. Because of the more complicated algorithm flow, this class does not allow as much freedom as previous GA or real–coded EA implementations. The example of PSO algorithm in FFEAT is in Algorithm 4.3.

Algorithm 4.3: PSO algorithm in FFEAT

```
import ffeat.pso as pso


fn = create_problem_function()

alg = pso.PSO(
    # Randomly initialize position of 100 particles
    pso.initialization.Uniform(100, -5.0, 5.0, 40),
    # Randomly initialize velocities
    pso.initialization.Uniform(100, -1.0, 1.0, 40),
    # Pass the evaluation function
    pso.evaluation.Evaluation(fn),
    # Specify random neighborhood with 3 neighbors
    pso.neighborhood.Random(3),
    # Use SPSO2006 velocity update rule
    pso.update.PSO2006(
        inertia=0.8,
        local_c=1.5,
        global_c=1.5
    ),
    # Clip particles position into range
    clip_position=pso.clip.Position(-5,5),
    # Clip particles velocity
    clip_velocity=pso.clip.VelocityNorm(2.0),
    # Specify number of iterations
    iterations=200
)
alg() # run the PSO algorithm
```

5. Evaluation

I will begin by introducing test problems, against which I will evaluate the implementation. The subsequent section describes the hardware specification of the machines I tested the implementation on. The discussion about the results is in the last section with selected measurements. You can see complete results in Appendix C. The hyperparameters of the experiments are in Appendix B.

5.1 Problems Description

For the Genetic Algorithms experiments I used SAT and 3SAT problems with various number of literals and clauses. The fitness function sums the number of unsatisfied (for minimization problem) or satisfied (for maximization problem) clauses. I implemented the fitness evaluation vectorized and fully in PyTorch, so the whole population is evaluated at once. I generated a new problem for every new experiment and all the problems were satisfiable. I achieved that by generating the assignment first and then generating the desired number of clauses. At least one literal in each clause matches the value with his corresponding counterpart in the generated assignment. Because I am more concerned with the running time of the algorithm rather than its performance, the satisfiability of the formulas does not make a big impact on the experiments.

The problem formula is kept as a matrix, where rows correspond to clauses and columns to indices of the literals. For a negative literal, the index is negative. This coding is efficient for 3SAT problem, but I run into a problem with generic SAT problem with a diverse number of literals in the clause. Because the tensors in PyTorch need to be aligned in each dimension, I transform the generic SAT problem into k-SAT problem, where k is equal to the length of the longest clause. The shorter clauses duplicate their first literal, so their truth evaluation does not change, and all the clause has exactly k literals. This may lead to a considerable inefficiency if the length variance between clauses is high. It may be worth to divide long clauses into smaller ones to reduce the overall size of the tensor and save some memory and computation. I believe this is not a serious obstacle for a broader application of the implementation.

The implementation processes files in the *DIMACS CNF* format, which is the general and standard file format to define Boolean expression written in conjunctive normal form [**?**]. The evaluation and parser implementation, with the script to generates the problems, are in the attachment.

For the Particle Swarm Optimization and real–coded evolutionary algorithms I used well–established Comparing Continuous Optimizers (COCO) Black–Box Optimization Benchmarking (BBOB) test suite [**?**]. The suite consists of 24 functions in 5 groups with increasing difficulty. The groups vary in their separability, conditioning, unimodality, and global structure. It would not be feasible for me to measure the algorithms on all of them. Furthermore, as I am more interested in running time of the algorithm, there is no need to evaluate the implementation on all of them. The shifts in the running time would be caused primarily by the speed of the function evaluation rather than the algorithm itself.

The functions are randomly shifted in the search space. The $\mathbf{x}^{opt}$ specifies the position of the optimum. The optimum is always kept in the $[-5, 5]$ interval in each dimension. Moreover, to eliminate the dependency of the algorithm on the absolute value of the function, it is shifted by the f_{opt} value sampled from the Cauchy distribution with zero mean and scale equal 100. This makes it difficult to directly use proportional–based selection operators because the optimum value differs each run and may be even negative. The algorithm does not know the $\mathbf{x}^{opt}$ and f_{opt} values, and these are only used for the measurements. Except $\mathbf{x}^{opt}$ and f_{opt}, the functions may have other parameters (typically rotation matrices $\mathbf{R}$, $\mathbf{Q}$, diagonal scaling matrix Λ), that I do not describe here. Moreover, the functions may use functions to break symmetry (T^{β}_{asy}), penalize the parameters (f_{pen}), and add noise (T_{osz}). The exact definition is in the BBOB specification. These parameters are initialized randomly before each run. Lastly, the functions allow to specify their dimension D at runtime, and the parameters are initialized accordingly. Therefore, it is easy to evaluate the algorithm on the same function with a different number of dimensions and hence on the problem with different difficulty.

I chose the following functions to evaluate the implementation. Not all the aspects of the algorithms were tested on all of these functions.

- Function f_1 – sphere function, that is unimodal, highly symmetric, rotationally and scale invariant. Sphere function is probably the simplest one and can be easily solved using local search techniques.

$$\begin{aligned} f_1(\mathbf{x}) &= \|\mathbf{z}\|^2 + f_{opt} \\ \mathbf{z} &= \mathbf{x} - \mathbf{x}^{opt} \end{aligned}$$

- Function f_7 – step ellipsoidal function. This function is unimodal, non–separable and has low conditioning. Because of the function step nature, it has many plateaus with zero gradient, so the gradient–based methods would not be very successful. Nevertheless, the function still exhibit ellipsoidal shape.

$$\begin{aligned} f_7(\mathbf{x}) &= 0.1 \max\left(\frac{|\hat{z}_1|}{10^4}, \sum_{i=0}^{D} 10^{2\frac{i-1}{D-i}} z_i^2\right) + f_{pen}(\mathbf{x}) + f_{opt} \\ \hat{\mathbf{z}} &= \Lambda^{10}\mathbf{R}\left(\mathbf{x} - \mathbf{x}^{opt}\right) \\ \tilde{z}_i &= \begin{cases} \lceil 0.5 + \hat{z}_i \rceil & \text{if } \hat{z}_i > 0.5 \\ \lceil 0.5 + 10\hat{z}_i \rceil / 10 & \text{otherwise} \end{cases} \\ \mathbf{z} &= \mathbf{Q}\tilde{\mathbf{z}} \end{aligned}$$

- Function f_{15} – Rastrigin function, that is non–separable, have roughly 10^D local optima, low conditioning and global amplitude larger than local one. The function is highly multimodal and is not regular nor symmetric.

$$\begin{aligned} f_{15} &= 10\left(D - \sum_{i=1}^{D}\cos(2\pi z_i)\right) + \|\mathbf{z}\|^2 + f_{opt} \\ \mathbf{z} &= \mathbf{R}\Lambda^{10}\mathbf{Q}T^{0.2}_{asy}\left(T_{osz}\left(\mathbf{R}\left(\mathbf{z} - \mathbf{x}^{opt}\right)\right)\right) \end{aligned}$$

- Function f_{19} – composite Griewank–Rosenbrock function. This function is highly multimodal and noisy.

$$
\begin{aligned}
f_{19}(\mathbf{x}) &= \frac{10}{D-1}\sum_{i=1}^{D-1}\left(\frac{s_i}{4000} - \cos(s_i)\right) + 10 + f_{opt} \\
\mathbf{z} &= \max\left(1, \frac{\sqrt{D}}{8}\right)\mathbf{R}\mathbf{x} + 0.5 \\
s_i &= 100\left(z_i^2 - z_{i+1}\right)^2 + (z_i - 1)^2 \\
\mathbf{z}_{opt} &= \mathbf{1}
\end{aligned}
$$

- Function f_{22} – Gallagher's Gaussian 21–hi peaks function with 21 unrelated and random optima.

$$
\begin{aligned}
f_{22}(\mathbf{x}) &= T_{osz}\left(10 - \max_{i=1}^{21} w_i \exp\left(-\frac{1}{2D}(\mathbf{x} - \mathbf{y_i})^T \mathbf{R}^T \mathbf{C}_i \mathbf{R}(\mathbf{x} - \mathbf{y_i})\right)\right)^2 + \\
&\quad + f_{pen}(\mathbf{x}) + f_{opt} \\
w_i &= \begin{cases} 1.1 + 8\frac{i-2}{19} & \text{for } i = 2, \ldots, 21 \\ 10 & \text{for } i = 1 \end{cases} \\
\mathbf{C}_i &= \Lambda^{\alpha_i} / \sqrt[4]{\alpha_i} \\
\alpha_i &= \begin{cases} \alpha_i = 10^6 & \text{for } i = 1 \\ \alpha_i \in \left\{1000^{2\frac{j}{19}} | j = 0, \ldots, 19\right\} & \text{sampled randomly without replacement for } i \neq 1 \end{cases}
\end{aligned}
$$

- Function f_{24} – Lunacek bi–Rastrigin function. This function is highly multimodal and deceptive, because of the promising area with local optima.

$$
\begin{aligned}
f_{24}(\mathbf{x}) &= \min\left(\sum_{i=1}^{D}(\hat{x}_i - \mu_0)^2, dD + s\sum_{i=1}^{D}(\hat{x}_i - \mu_1)^2\right) + \\
&\quad + 10\left(D - \sum_{i=1}^{D}\cos(2\pi z_i)\right) + 10^4 f_{pen}(\mathbf{x}) \\
\hat{\mathbf{x}} &= 2\text{sign}\left(\mathbf{x}^{opt}\right) \bigotimes \mathbf{x} \\
\mathbf{x}^{opt} &= \mu_0 \mathbf{1}^+_- \\
\mathbf{z} &= \mathbf{Q}\Lambda^{100}\mathbf{R}(\hat{\mathbf{x}} - \mu_0\mathbf{1}) \\
\mu_0 &= 2.5, \mu_1 = -\sqrt{\frac{\mu_0^2 - d}{s}}, s = 1 - \frac{1}{2\sqrt{D+20} - 8.2}, d = 1
\end{aligned}
$$

I reimplemented all the BBOB functions in PyTorch. They are fully vectorized so that the whole population can be evaluated at once. The implementation of these functions is in the attachment and is also available from PyPI as the *BBOBtorch* package.

5.2 Hardware Specification

I run all my experiments in MetaCentrum. I used servers with specifications in Table 5.1a for workloads running on CPU. For GPU specialized tasks, I used servers with hardware specifications in Table 5.1b.

CPU	AMD EPYC™ 7452
RAM	256 GiB
Disk	2×4 TB HDD
Owner	Faculty of Science, Charles University

(a) Hardware specification for CPU experiments

CPU	Intel® Xeon® Gold 5218
RAM	192 GiB
Disk	4×240 GB SSD
GPU	nVidia Tesla T4
GPU memory	16GB
CUDA cores	2560
Tensor cores	320
Owner	CESNET

(b) Hardware specification for GPU experiments

Table 5.1: Hardware specification of servers the implementation was tested on

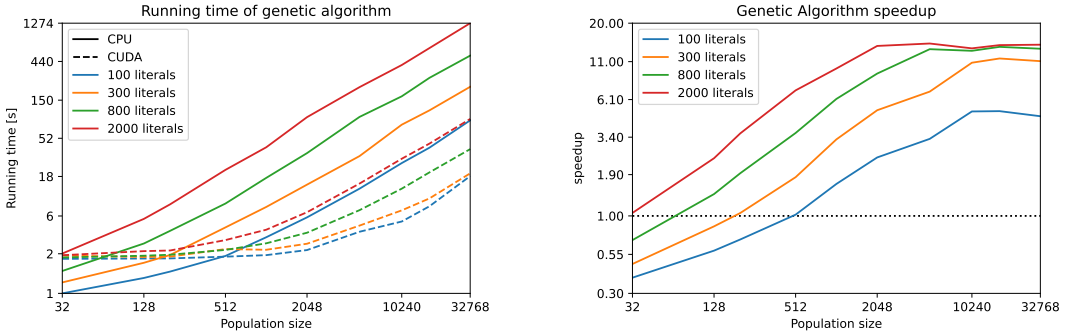
One drawback of using MetaCentrum is that the machines are shared amongst the academic community of the Czech Republic. I could not block the whole machine for a more extensive time because of the terms of use, moreover, it would not be morally right. I have done all the CPU workloads using tasks with eight cores and all the GPU workloads with two cores.

While some of the resources, for example GPU and RAM, are allocated exclusively to the running task, other, for example disk, network bandwidth, and CPU to some extent, are not. The CPU situation is further complicated by the Hyper–Threading technology. This technology duplicates one physical processor core into two logical ones. While the first may use the full utilization of the core, the second one uses the idle portion of it. All the processors currently installed in the MetaCentrum, including the AMD EPYC 7452 processor, use this technology. As the goal of this work is focused on the computation quantity rather than on the raw computation power, experiments done on this second logical core may have a significant impact on the execution time, especially for the CPU workloads. I did my best to mitigate this issue by allocating extra cores, running the experiments during unoccupied hours and on the same machine, so that no other user could interfere. Nevertheless, the measurements may still be noisy.

5.3 Results Discussion

I run all the experiments on the hardware specified above, and because the Evolutionary Algorithms are by their nature stochastic, I repeated each experiment a hundred times. The reported numbers are the mean of the given metric over these runs.

First CUDA call from the PyTorch runtime initialize the CUDA context and based on my measurements, it takes about 1.5s. The initialization is done only once and it could add bias to the first run of the algorithm. I decided not to take the delay into account and all the measurements in this work are without it. I would argue this delay is spread over the runs, and because of the stochastic nature of the EA, they should be executed multiple times in order to get reliable results.



(a) Running time of GA

(b) Speed up of GA on GPU

Figure 5.1: Genetic Algorithms evaluation

The experiments run for the population of sizes 32, 128, 200, 512, 1024, 2048, 5000, 10240, 16384, and 32768 in the most cases. These values play a role in all the experiments and, most of the time, are the quantity shown on the x axis. I show two types of results – running time with respect to population size and fitness progress with respect to generations. Note that the second named needs to be taken with a grain of salt. Bigger population has advantage of more fitness evaluations (up to three orders of magnitude) and, therefore, is in advantage. For practical application one needs to take into account not only the fitness, but also the running time of the algorithm.

Based on the experiments, the CUDA implementation seems to perform better from medium to big–sized problems and populations. That is not surprising, as the GPU is intended for vast computation and memory demands. For small problems and populations, the runtime stays constant up to a certain threshold, where the runtime starts to increase linearly with the problem size. This is clearly visible in Figure 5.1a for 3SAT problem with 100, 300, and 800 literals. In the case of 2000 literals, I would already classify the problem as medium-sized. The time for even the smallest population of 32 individuals is the same for both CPU and GPU implementation. Measurement of 3SAT problem with varying number of literals and clauses is depicted in Figure C.2 and the GPU implementation clearly outperform the CPU one over the whole domain. Note that I used a population size of 1000 individuals, which is advantageous for the GPU implementation.

The interesting observation is the constant run time of the GPU implementation for small populations and problems. In these cases, the GPU is not fully utilized, and the whole population can be processed truly in parallel. As the CUDA cores are less performant than CPU cores, the total running time is higher than of CPU. Moreover, the communication overhead between CPU and GPU plays a relevant role in the running time. The speedup of the algorithm running on the GPU is depicted in Figure 5.1b.

I compared the PyTorch implementation to C++ one in Figure 5.2. You can find the C++ implementation in the attachment. It used master–worker architecture, where the fitness evaluation is done in parallel using multiple cores, whereas the rest of the algorithm is sequential. Both implementations use the same hyperparameters specified in Appendix B. I compared the implementation using one and eight cores, along with the GPU implementation. You may notice that C++ implementation using one core outperforms PyTorch one by around 80%. Python

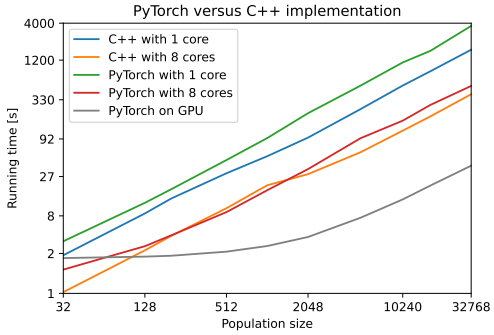


Figure 5.2: Running time of C and PyTorch implementation

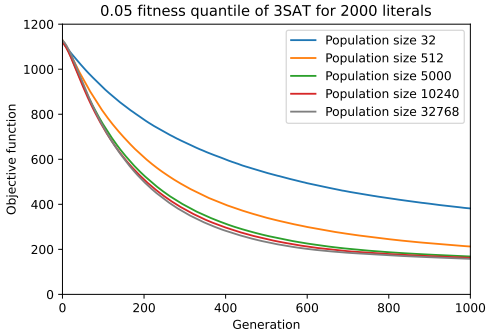


Figure 5.3: 0.05 quantile of fitness on 3SAT problem with 2000 literals

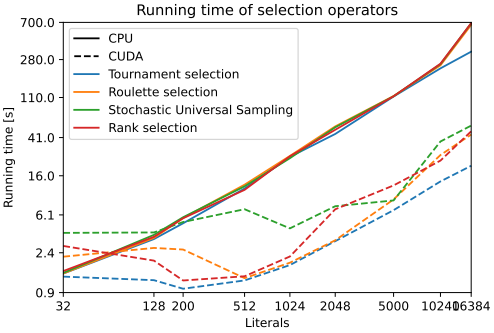


Figure 5.4: Running time of selection operators

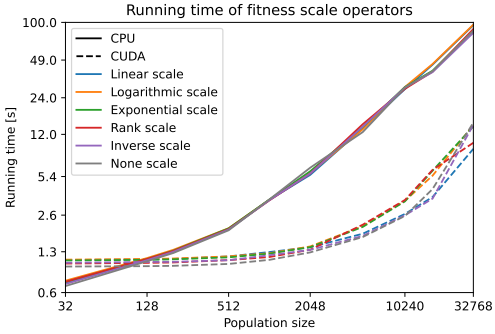


Figure 5.5: Running time of fitness scale operators for problem with 100 literals.

is an inherently slow language, and the additional overhead of calling native code from Python underlines it. Moreover, the PyTorch implementation is a bit complicated compared to the simple C++ implementation, which may also influence the running time. When I compared the implementations using eight cores, the difference is almost negligible (at most 20%). The PyTorch implementation took advantage of the parallelization of all the operators, and I believe an increased number of cores would further favor PyTorch implementation. The GPU implementation still outperforms the C++ implementation by order of magnitude on populations larger than five hundred individuals.

Of course, one would use a larger population only if it is advantageous. I measured the progress of fitness value over the generation, and these experiments are depicted in Figure C.3 (in Figure 5.3 for 2000 literals) and C.4 (with elitism). I measured the 0.05 quantile of the fitness function, and the algorithm converges gradually faster for populations of size 32, 512, and 5000. The big populations with 10240 and 32768 individuals do not make an impact until solving big enough 3SAT problem with 2000 and 5000 literals. Unfortunately, the improvement is not as significant as between the populations of size 512 and 5000. In my opinion, using that big population for this kind of problem is not worth it.

Running time of various selection operators is in Figure 5.4 (for 3SAT problem). Note that this is one of the more noisy experiments, although I repeat the experiment multiple times. For the CPU implementation, the fitness evaluation is taking most of the time and the impact of the selection operator is scant. On

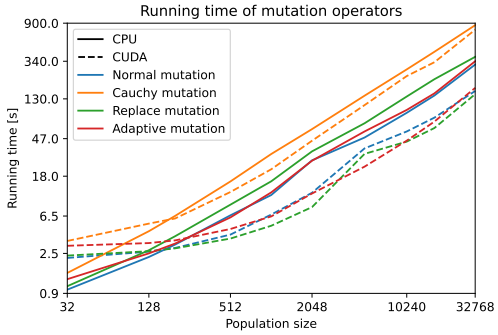


Figure 5.6: Running time of mutation operators for BBOB function f_{19} with 384 dimensions

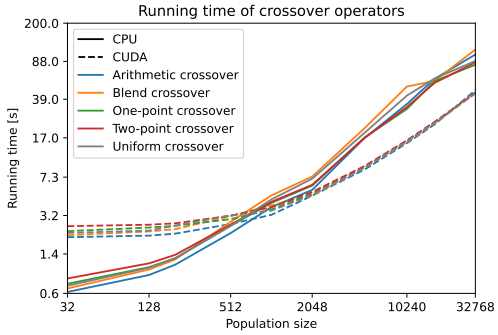


Figure 5.7: Running time of crossover operators for BBOB function f_{19} with 128 dimensions

the GPU is the situation different, and the selection operators seem to have an order of magnitude difference. The fastest one is the tournament selection, which is not surprising, as it does not require sorting or search in the fitness array. On the other hand, the SUS seems to be the slowest one, especially for smaller populations. For bigger populations of the size greater than 5000 the performance of roulette, SUS and rank selection looks very similar.

Lastly, I measured the impact of fitness scaling operators on the run time of the GA, and these results are in Figure 5.5 (experiments for 1000 literals are in Figure C.5). The algorithm used tournament selection, so the different scale operators did not influence the algorithm running time. Similarly to the selection, the fitness evaluation took the majority of the time, and the execution of the scale function does not make any impact when executed on CPU (including the rank scale operator). The situation with GPU was very similar except for a small problem (100 literals and 450 clauses), where logarithmic, exponential, and rank scaling took slightly more time.

I tested mutation operators on real–coded evolutionary algorithm with uniform crossover and tournament selection; the results for BBOB function f_{19} with 384 dimensions is in Figure 5.6 (see Figure C.8 for all the experiments). The fitness of normal and adaptive step mutations is in Figure 5.8 (all the experiments are in Figure C.9). Both CPU and GPU implementations were slowest using Cauchy mutation. The reason is that the sampling from the Cauchy distribution lacks efficient implementation. The fastest was normal mutation with replacement mutation in some GPU cases (on CPU the replacement mutation was always a bit slower than the normal mutation). The adaptive step mutation was somewhere in between because of the fitness comparison overhead. The replacement mutation is interesting, as it outperformed normal mutation on some GPU cases, but is steadily slower on CPU. I would say sampling a random value within an interval would be faster than sampling from the normal distribution and I cannot fully explain this.

From a convergence point of view, the complete experiments are shown in Figure C.9. The best performant mutation is the normal mutation, and as it is the fastest one, I would recommend it. Unfortunately, the various mutation operators do not take advantage of a bigger population, and except for population size 32, the performance of populations with sizes 10240 and 32768 is the same.

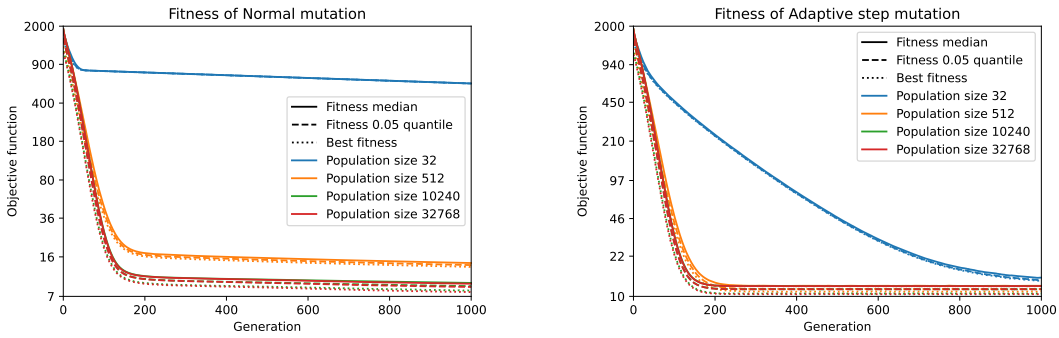


Figure 5.8: Fitness of normal and adaptive step mutation operators for BBOB function f_{19} with 384 dimensions

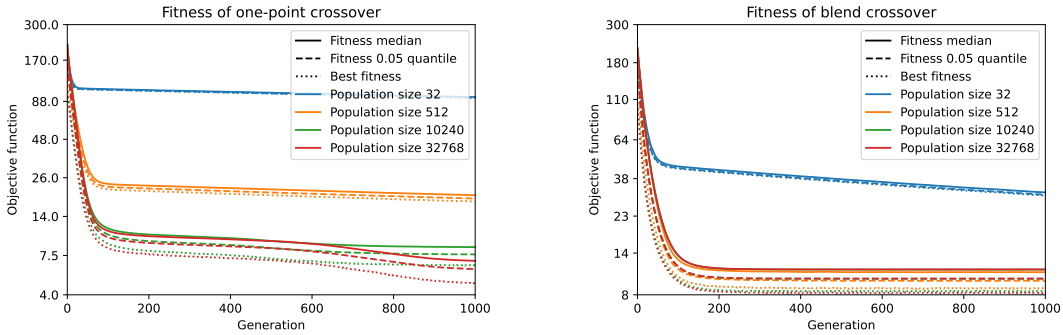


Figure 5.9: Fitness of one–step and blend crossover operators for BBOB function f_{19} with 128 dimensions

The algorithm converged a little faster with 10240 individuals instead of 512, but only for a problem with 384 dimensions (see adaptive step mutation in Figure 5.8) and the difference is negligible. The only exceptions worth mentioning is the normal and Cauchy mutation for a problem with 24 dimensions, where a larger population helped finding better optima. Bigger population also helped normal mutation in 384 dimensions (see Figure 5.8), but this trend does not hold for population with 32768 individuals. Lastly, you may notice better convergence properties of adaptive step mutation in contrast to normal or Cauchy mutation.

The crossover operators were tested on real–coded evolutionary algorithm, and the running times are in Figure 5.7 (in Figure C.10 are all the experiments) along with their fitness values in Figure 5.9 (complete experiments in Figure C.11). The running times were of the same order for all of them. The slowest on the CPU was the blend crossover, while the arithmetic crossover was the fastest one. The slowest on the GPU was the two–points crossover because of the complicated creation of the mask, as discussed in the Chapter 4. The fastest operator for small populations was the arithmetic crossover with the blend crossover. The running time for big populations on GPU was almost identical for all the operators. Unlike mutation operators, some of the crossover operators take advantage of the larger population, as shown in Figure 5.9 and C.11 – for example the one–point and two–point crossovers. On the other hand, blend crossover performed best using 512 individuals, and a bigger population decreased its performance.

Lastly, the crossover schema experiments are depicted in Figure 5.10 (all ex-

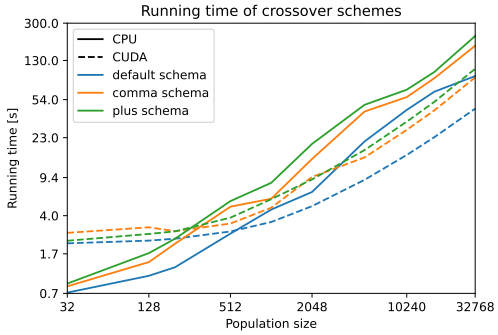


Figure 5.10: Running time of crossover schemes for BBOB function f_{19} with 128 dimensions

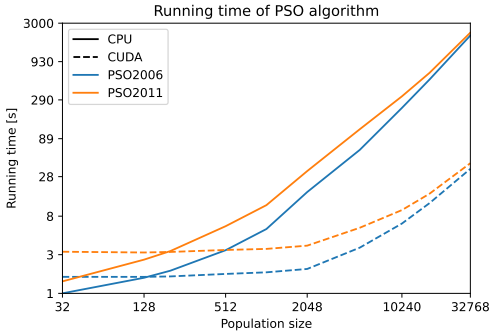


Figure 5.11: Running time of PSO algorithm for BBOB function f_{19} with 384 dimensions.

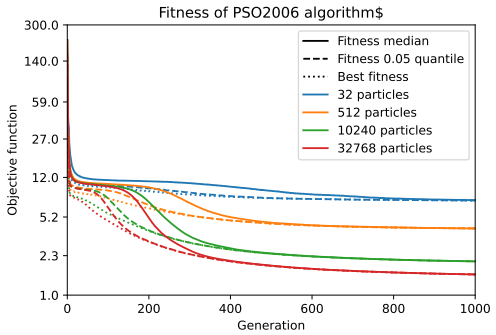


Figure 5.12: Fitness of PSO algorithm for BBOB function f_{19} with 128 dimensions

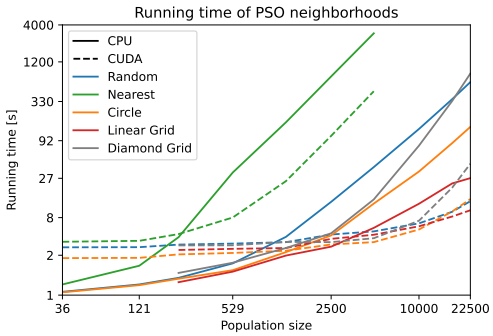


Figure 5.13: Running time of PSO algorithm with various neighborhood types for BBOB function f_{19} with 128 dimensions.

periments are in Figure C.12). There is no surprise that the default schema (that is, the offspring replace their parents in the population) is the fastest one because it does not require extra memory allocation, as discussed in Section 4.1. It is followed by comma schema, which is almost two times slower. The plus schema is the slowest and runs around 2.3× slower than the default schema.

Finally, the running time of SPSO–2006 and SPSO–2011 algorithms are shown in Figure 5.11 (see Figures C.13 and C.17 for more details). The PSO algorithm shows better speedup than GA and real–coded evolutionary algorithms, even for problems with 384 dimensions. The running times for problem with 6, 32, and 128 dimensions is almost identical except the BBOB function f_{22}. This function has the most complicated evaluation and it takes great portion of the running time. The graphs showing algorithm fitness for BBOB function with 128 dimensions is in Figure 5.12 (more detailed experiments are in Figure C.14 for SPSO–2006, and in Figure C.18 for SPSO–2011). Unlike previous experiments, all the problems take advantage of the larger population, as is clearly visible in the figures. I would say the PSO algorithm has the biggest potential to run on GPU.

The PSO neighborhoods experiments are depicted in Figure 5.13 (experiments for other BBOB problems are in Figure C.15), and the fitness is in Figure C.16. The random and circle neighborhoods are the fastest to evaluate on GPU, where the circle one is a bit faster for small populations. The circle neighborhood is

	Number of experiments	**Running time**
Genetic Algorithms	71 068	999.9h
Real–coded evolutionary algorithms	142 866	1136.8h
Particle Swarm Optimization	195 419	1264.7h
Hyperparameter search	7 476	49.5h
Testing runs	5 147	53.5h
Failed runs	30 578	136.6h
CPU runs	231 352	2567.6h
GPU runs	221 202	1073.4h
CPUdays according to MetaCentrum		2149.2h
Total	452 554	3641.0h

Table 5.2: Experiments running times

static and generated only once, whereas the random neighborhood changes with every iteration. The random neighborhood is the slowest on the CPU (except the nearest one) precisely for this reason. The grid–based neighborhoods are as fast as circle neighborhood for smaller populations, but for bigger populations the evaluation takes as long as random neighborhood. The cause is their substantial size in contrast to the circle one. The exception is the linear grid, which is the smallest of all the grid–based neighborhoods. Note that as circle and grid–based neighborhoods are static neighborhoods, their running time for the same neighborhood size should be equal. The nearest neighborhood is extensively slower than any other. This is expected because it needs to measure the distance between every pair of particles, which is very costly even on GPU. The neighborhood evaluation is still more than six times faster on the GPU for swarms larger than 500 particles.

The overall statistics about the experiments are given in Table 5.2, including the number of experiments and the total running time. Note that some experiments failed, and I repeated some experiments to get more accurate measurements. The numbers, therefore, do not align to hundreds.

6. Conclusion

This work summarizes current knowledge of Evolutionary Algorithms and gives a comprehensive overview of this research field. It discusses the well–known evolutionary operators and gives their detailed analysis from the implementation point of view, along with their desired properties. The work further describes the three most common evolutionary algorithms – Genetic Algorithms, Real–Coded Evolutionary Algorithms, and Particle Swarm Optimization Algorithms. Each of them is analyzed, and their representative operators are described and examined.

The work further discusses the differences between CPU and GPU and their architectures. I show that while the GPU has its application in various fields mentioned in the work, the mindset for programming on GPU is considerably different and unique. I present the OpenCL and CUDA technologies, and introduce the terminology used in GPU programming. The work shows the elements of CUDA programming, with a particular focus on a performance issue that can arise.

I then present the PyTorch library for the Python programming language, describe its architecture and functionality. The work put into connection the PyTorch library and CUDA programming discussed before. Follows discussion about possible ways to parallelize Evolutionary Algorithms both on CPU and GPU. The work focuses on different individual encoding, parallelization granularity, and evaluation architectures.

Follows presentation of the Framework For Evolution Algorithms in Torch (FFEAT), my contribution to this area. I discuss in detail its functionality, implementation decisions, architecture, and the proposed workflow. The work gives extra space to challenging parts of the implementation, like parental sampling and crossover schemes. Some of these problems arise from the limitations of the PyTorch library and the CUDA programming in general; these aspects are discussed. Implementation of some operators is then shown in the text and I discuss possible ways to expand the library. Overall, I show that the proposed implementation is easy to extend, operators are written in a readable manner and runs efficiently on GPU.

The implementation is subjected to extensive testing on 3SAT problem and BBOB test suite. The experiments show the advantage of using CUDA implementation for medium and big–sized populations and problems. The implementation shows an order of magnitude speedup when evaluated on the GPU rather than on CPU. Moreover, the experiments show the advantage of a bigger population on some problems while leading to premature convergence on others. I discuss this phenomenon and explain the behavior of algorithms.

Finally, I compare the proposed implementation against the native C++ implementation on the 3SAT problem. It shows a noticeable slowdown on one core machine but gets faster as the number of cores increases. The proposed implementation makes use of the parallelized operators implementation and becomes dominant. The comparison with C++ implementation confirms the superiority of CUDA implementation for medium and big–sized populations.

This work dealt with GA, real–coded evolutionary algorithms, and PSO algorithms. There are other classes of algorithms that the implementation was not

tested on. Further work could expand the library with these algorithms. I believe the CMA–ES would greatly benefit of CUDA implementation. Benchmark the implementation on permutation–based algorithms may seems interesting as well. Furthermore, all the presented problems have been vectorized, and the GPU implementation could evaluate the whole population at once. It may be interesting to explore the possibility of evaluating each individual separately, taking away the major advantage of the GPU implementation. This may be practical for complicated fitness functions. Future work could explore this possibility and evaluate whether GPU implementation keeps its advantages.

List of Figures

List of Tables

List of Algorithms

A. Selected Operators Implementation

Algorithm A.1: Tournament selection implementation

```
from typing import Tuple, Any, Dict, Union, Callable
import torch as t
from ffeat import Pipe
from ffeat.utils._parental_sampling import randint

_IFU = Union[int, float]

class Tournament(Pipe):
  def __init__(self, num_select=None, maximization=
  ↪ False, parents=2, parental_sampling=randint):
    self._num_select = self._handle_parameter(
  ↪ num_select)
    self._maximization=maximization
    self._parents = parents
    self._parental_sampling = parental_sampling

  def __call__(self, fitnesses, population, *args, **
  ↪ kwargs):
    originally = len(population)
    to_select = self._num_select(fitnesses,
  ↪ population, *args, **kwargs)

    indices = self._parental_sampling(originally,
  ↪ to_select, self._parents, fitnesses.device)
    operation = t.argmax if self._maximization else t
  ↪ .argmin
    best_indices = operation(fitnesses[indices], dim
  ↪ =1)
    selected = population[indices[range(to_select),
  ↪ best_indices]]

    return (selected, *args), kwargs
```

Algorithm A.2: One–point crossover operator

```
from typing import Tuple, Any, Dict, Union
import torch as t
from ffeat import Pipe
from ._Shared import _CommonCrossover
from ffeat.utils._parental_sampling import randint

class OnePoint1D(Pipe, _CommonCrossover):
  def __init__(self, offsprings, replace_parents=True
  ↪ , in_place=True, discard_parents=False,
  ↪ parental_sampling=randint):
    _Shared.__init__(self, offsprings,
  ↪ replace_parents, in_place, discard_parents)
    self._parental_sampling = parental_sampling

  def __call__(self, population, *args, **kwargs):
    ptp = population.dtype if population.dtype != t.
  ↪ bool else t.uint8
    dev = population.device
    dim = population.shape[1]
    num_crossovers = self._offsprings // 2
    crossover_indices = t.randint(1, dim, size=(
  ↪ num_crossovers,), dtype=t.long, device=dev)
    parents_indices = self._parental_sampling(len(
  ↪ population), num_crossovers, 2, dev).T
    children = t.zeros((num_crossovers, dim), dtype=
  ↪ ptp, device=dev)
    position_mask = t.arange(dim, device=dev).
  ↪ as_strided((num_crossovers,dim), (0,1))
    lpos = (position_mask < crossover_indices[:, None
  ↪ ]).type(t.int8)
    children[:num_crossovers].add_(population[
  ↪ parents_indices[0]] * lpos)
    children[num_crossovers:].add_(population[
  ↪ parents_indices[1]] * lpos)
    rpos = t.logical_not(lpos, out=lpos)
    children[:num_crossovers].add_(population[
  ↪ parents_indices[1]] * rpos)
    children[num_crossovers:].add_(population[
  ↪ parents_indices[0]] * lpos)
    children = children.to(population.dtype)
    pop = self._handle_pop(population, children,
  ↪ parents_indices)
    return (pop, *args), kwargs
```

Algorithm A.3: Elitism operator implementation

```
from typing import Tuple, Any, Dict, Union, Callable
import torch as t
from ffeat import Pipe, flow

class Elitism(Pipe):
  def __init__(self,
         num_elites,
         *following_steps,
         maximization=False):
    self._num_elites = self._handle_parameter(
  ↪ num_elites)
    self._maximization = maximization
    self.__follow = flow.Sequence(*following_steps)

  def __call__(self, fitnesses, population, *args, **
  ↪ kwargs):
    to_select = self._num_elites(fitnesses,
  ↪ population, *args, **kwargs)
    elites_indices = t.topk(fitnesses, to_select,
  ↪ largest=self._maximization)[1]
    elites = t.clone(population[elites_indices])
    (population, *args), kargs = self.__follow(
  ↪ fitnesses, population, *args, **kwargs)
    population[elites_indices] = elites
    return (population, *args), kwargs
```

B. Hyperparameters

Here I present the hyperparameters used for experiments in the following chapter. I run a rough hyperparameter search to use reasonably good enough parameters, and the algorithm does not diverge. As the goal of this thesis is to evaluate the execution time of the algorithm rather than its quality or performance, these parameters may not be optimal for the given problem at hand. I recommend the available literature for a more appropriate hyperparameter selection.

Parameter	**Value**
Selection	Tournament
Number of parents	2
Crossover	Uniform
Probability to inherit gene from 1st parent	40%
Offspring	80%
Replace mutation	
Mutation rate	20%
Position clipping	no
Normal mutation	
Mutation rate	20%
Standard deviation	0.005
Position clipping	no
Cauchy mutation	
Mutation rate	20%
Scale	0.003
Position clipping	yes
Adaptive step mutation	
Starting deviation	0.01
Deviation increase	1.3
Deviation decrease	0.2
Fraction of better to increase deviation	30%
Minimum deviation	0.00001
Maximum deviation	1.0
Position clipping	no

Table B.1: Hyperparameters of mutation experiments

Parameter	Value
Selection	Tournament
Number of parents	2
Crossover	One–point
Mutated individuals	40%
Mutation	Bit–Flip
Mutated individuals	60%
Mutation probability	0.1%

Table B.2: Genetic Algorithms hyperparameters

Parameter	Value
Selection	Tournament
Number of parents	2
Mutation	Normal
Mutation rate	20%
Standard deviation	0.005
Uniform crossover	
Offspring	80%
Probability to inherit gene from 1st parent	40%
One–point crossover	
Offsprign	80%
Two–point crossover	
Offsprign	80%
Arithmetic crossover	
Offsprign	40%
Blend crossover	
Offsprign	70%
α	0.5

Table B.3: Hyperparameters of crossover experiments

Parameter	Value
Selection	Tournament
Number of parents	2
Crossover	Uniform
Probability to inherit gene from 1st parent	40%
Mutation	Normal
Mutation rate	20%
Standard deviation	0.005
Standard mutation	
Offspring	80%
Plus schema	
Offspring	150%
Comma schema	
Offspring	200%

Table B.4: Hyperparameters of crossover schemes experiments

Parameter		**Value**
Cognitive acceleration coefficient	c_l	1.5
Social acceleration coefficient	c_g	1.5
Inertia weight	ω	0.8
Velocity clip value		2

Neighborhood	**Relative size**
Circle	0.26
Grid	
Linear	0.02
Compact	0.05
Diamond	0.06
Nearest neighbors	0.13
Random	
PSO2006	0.13
PSO2011	0.2

Table B.5: Particle Swarm Optimization hyperparameters

C. Results

All the algorithms were run with hyperparameters given in appendix B. Each configuration ran a hundred times and the plotted values are mean of the specified metric. Results in higher resolution are in the attachment.

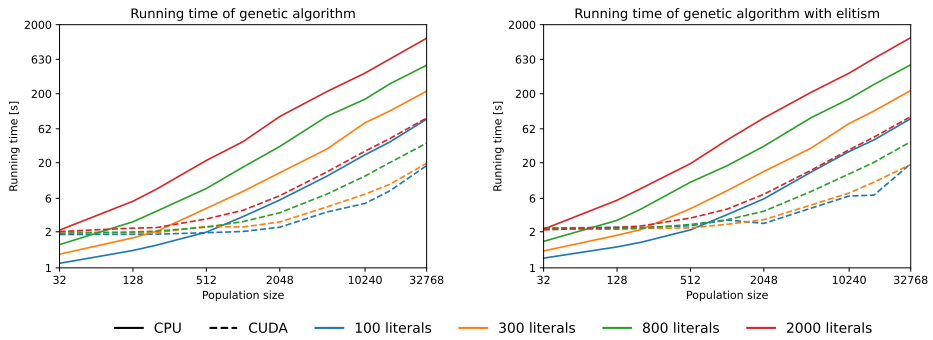


Figure C.1: Running time of Genetic Algorithms with and without elitism for 3SAT problem. All the formulas have been satisfiable and had 4.5 times more clauses than literals. The algorithm run for 1000 generations.

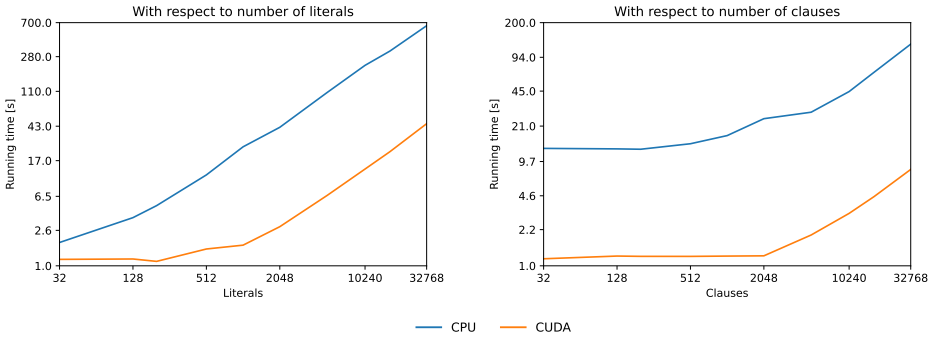


Figure C.2: Running time of Genetic Algorithms for 3SAT problem with respect to problem size. On the left with respect to number of literals with 4.5 times the number of clauses, and with respect to number of clauses for 2000 literals on the right. All the formulas have been satisfiable and measurements are done for the population size equal 1000. The algorithm run for 1000 generations.

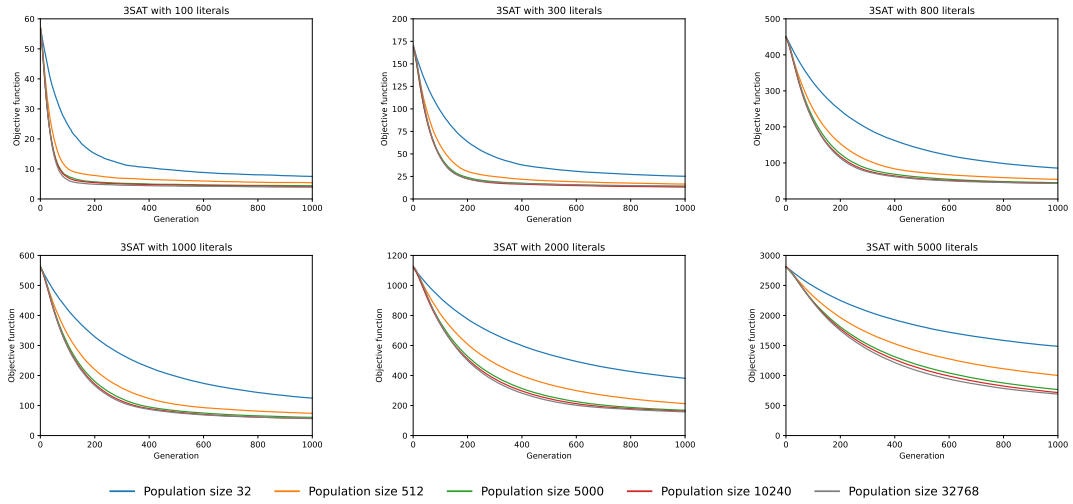


Figure C.3: Fitness of Genetic Algorithms on 3SAT problem. Algorithm run for maximum of 1000 generations and had 4.5 times number of clauses than the literals. The graphs show 0.05 quantile of the fitness value in the population.

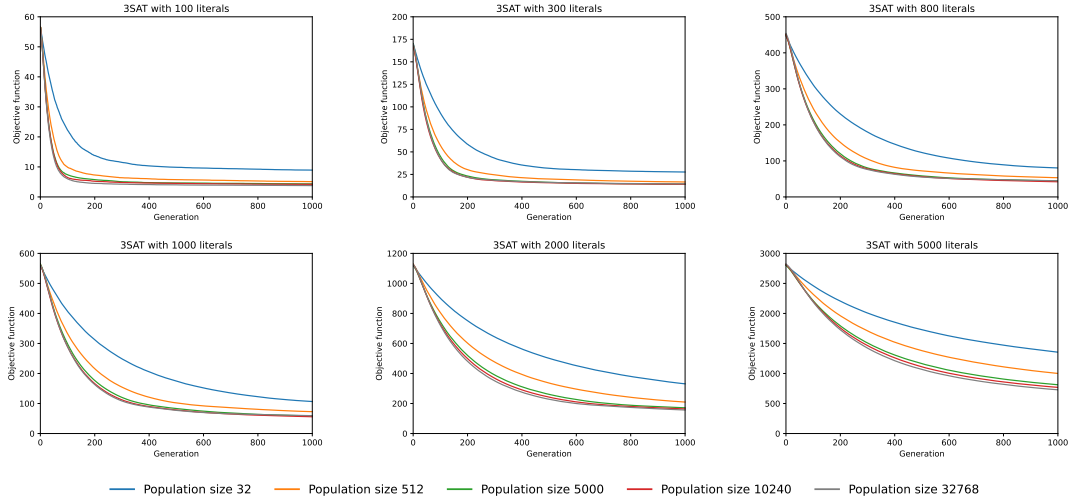


Figure C.4: Fitness of Genetic Algorithms with elitism on 3SAT problem. Algorithm run for maximum of 1000 generations and had 4.5 times number of clauses than the literals. The graphs show 0.05 quantile of the fitness value in the population.

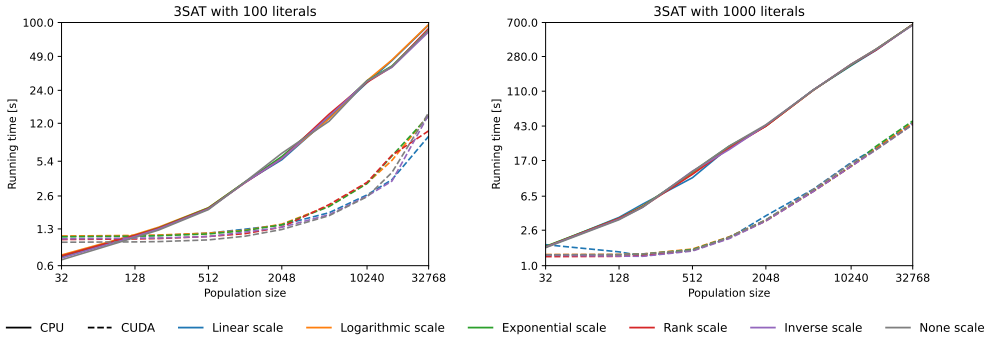


Figure C.5: Running time of Genetic Algorithms with various fitness scale operators. The algorithm were solving 3SAT problem with 100 (left) resp. 1000 (right) literals and 450, resp. 4500 clauses. The algorithm run for 1000 generations.

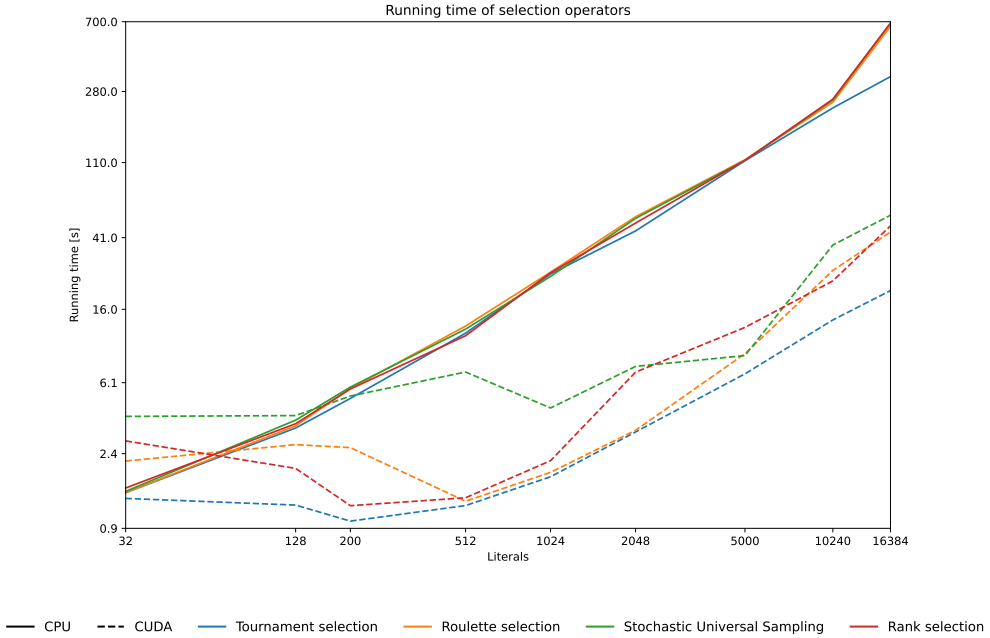


Figure C.6: Running time of Genetic Algorithms with various selection operators. The algorithm were solving 3SAT problem with 1000 literals and 4500 clauses, and run for 1000 generations.

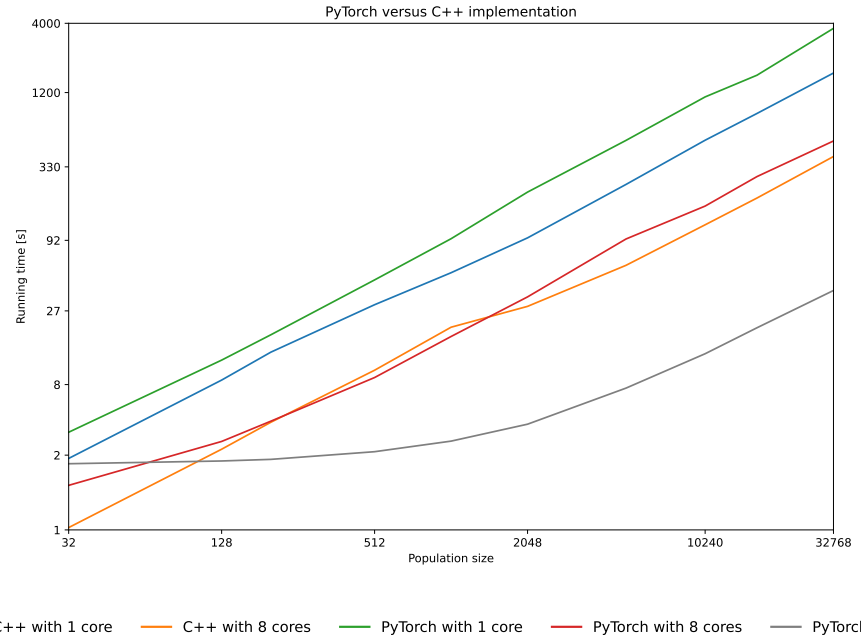


Figure C.7: Running time of Genetic Algorithms implemented in PyTorch and C++. The algorithm were solving 3SAT problem with 800 literals and 3600 clauses, and run for 1000 generations. The C++ implementation used the master–worker approach to evaluate fitness function in parallel.

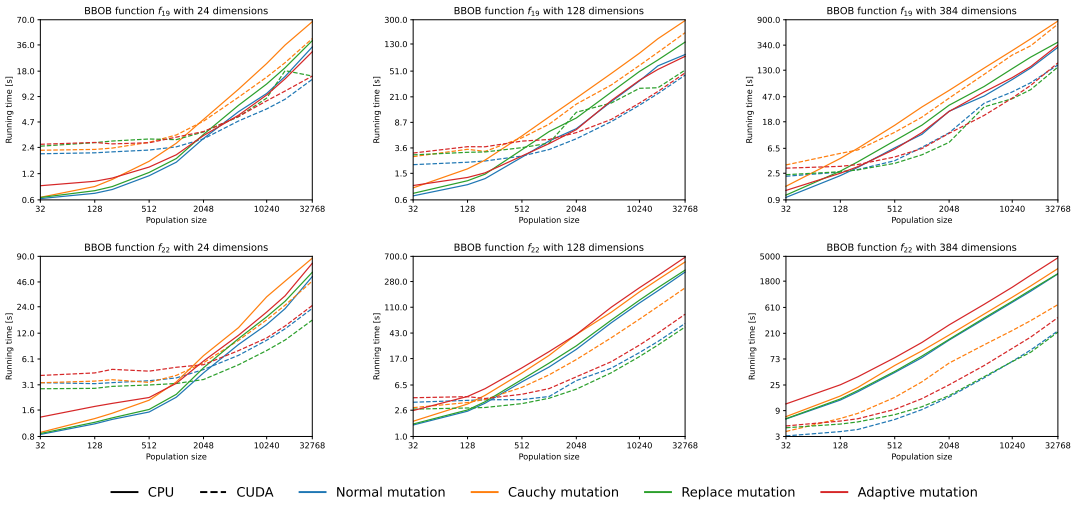


Figure C.8: Running time of real–coded evolutionary algorithm with various mutation operators. I chose to measure only BBOB functions f_{19} and f_{22}. Algorithm used hyperparameters specified in Table B.1 and run for 1000 generations.

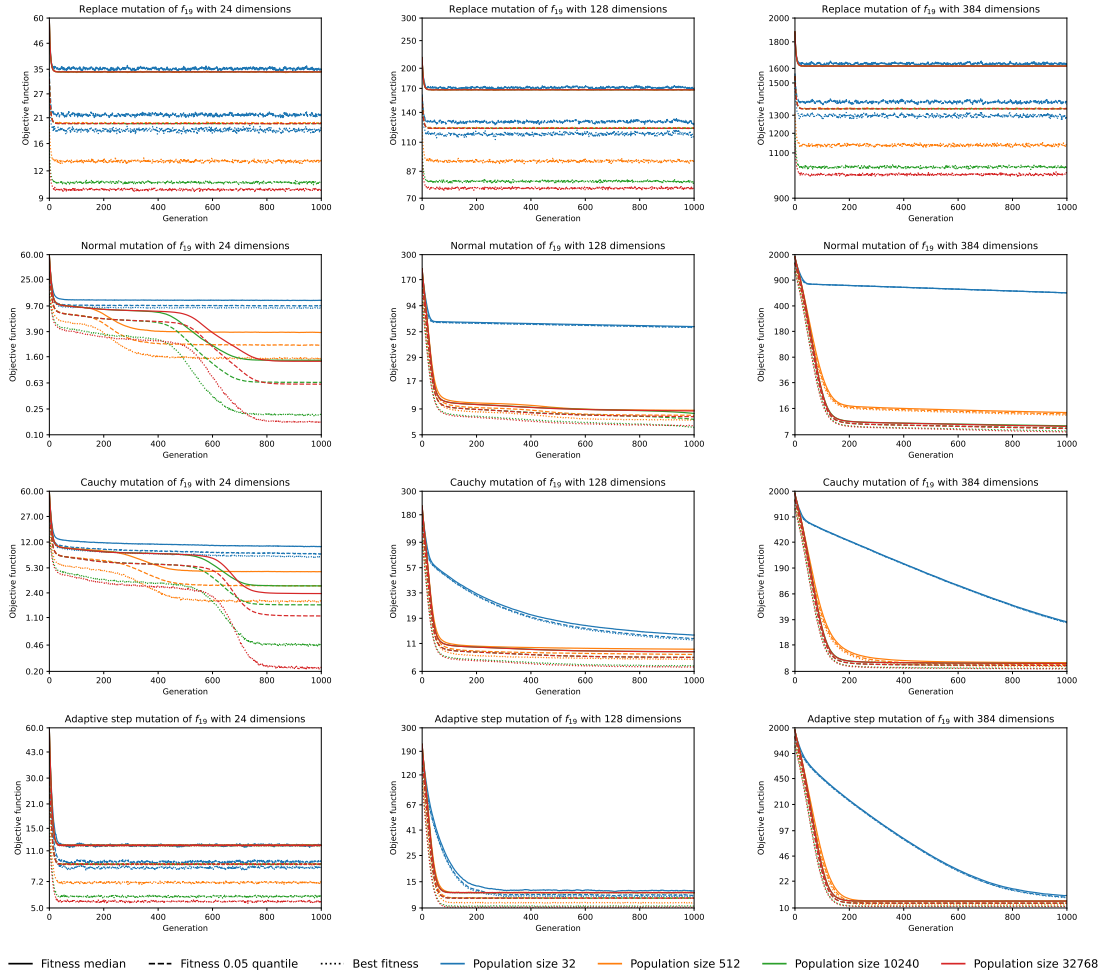


Figure C.9: Fitness of real–coded evolutionary algorithm with various mutation operators. Measured on BBOB functions f_{19} and f_{24} (available in attachment). Algorithm used hyperparameters specified in Table B.1.

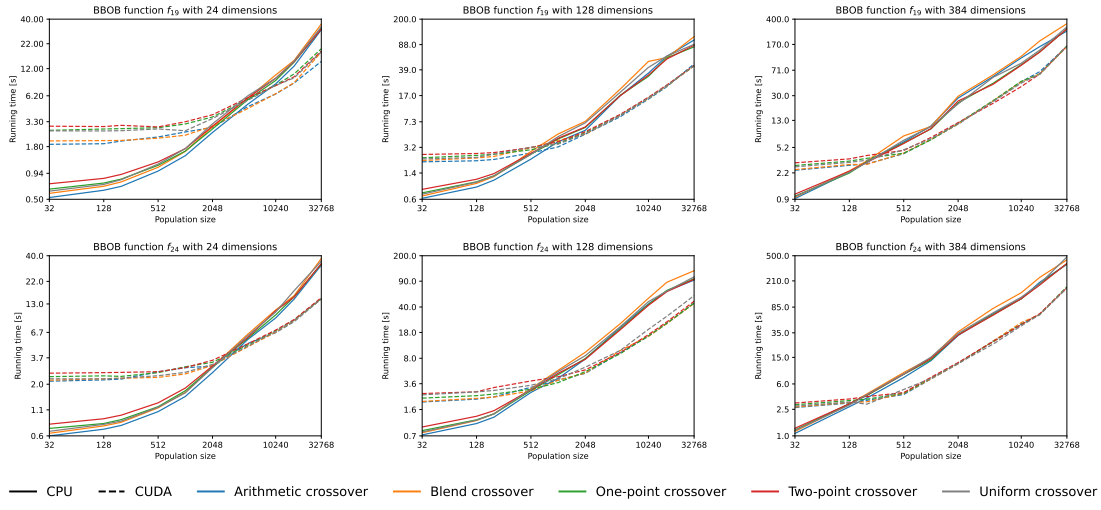


Figure C.10: Running time of real–coded evolutionary algorithm with various crossover operators. I chose to measure only BBOB functions f_{19} and f_{24}. Algorithm used hyperparameters specified in Table B.3 and run for 1000 generations.

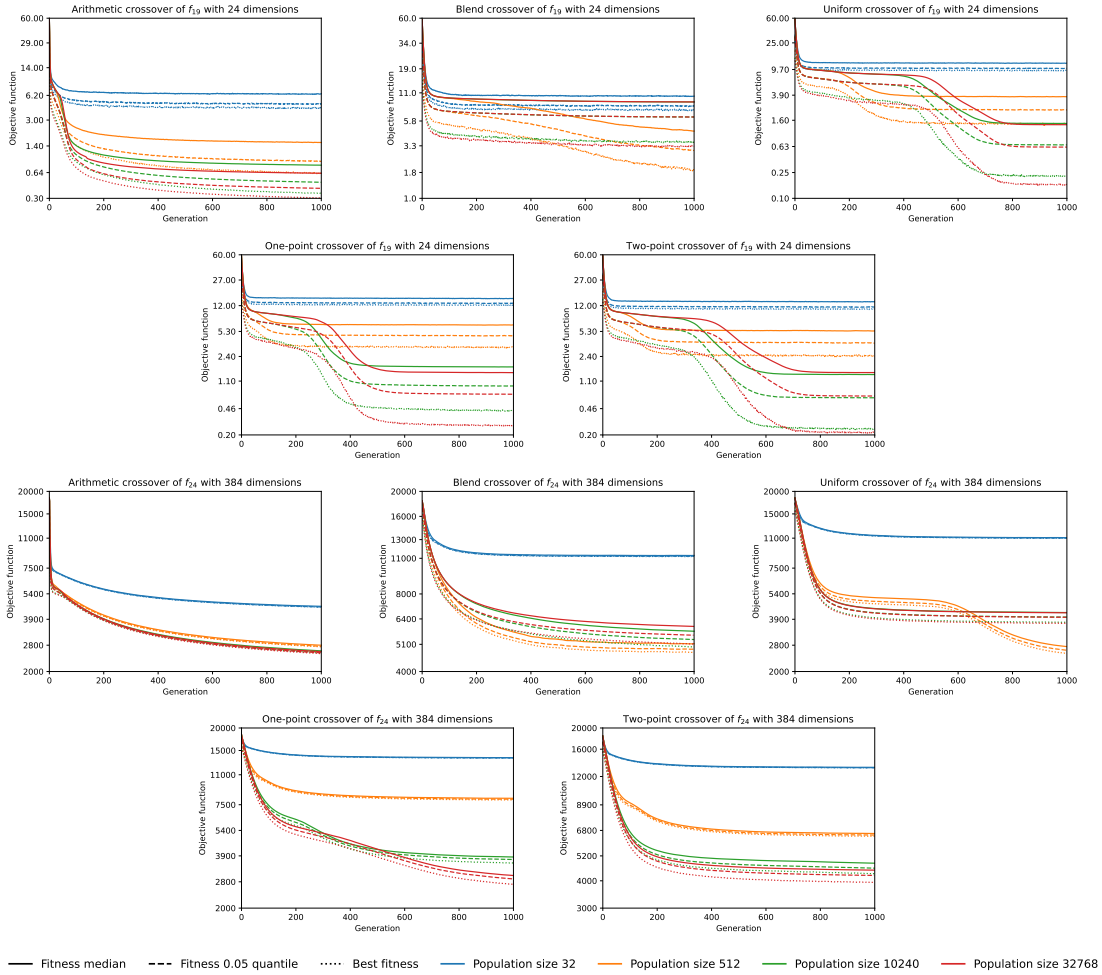


Figure C.11: Fitness of real–coded evolutionary algorithm with various crossover operators. Measured only on BBOB functions f_{19} and f_{24} with dimensions 24, 128, and 384. Algorithm used hyperparameters specified in Table B.3. I present only function f_{19} with 24 dimensions and function f_{24} with 384 dimensions, rest of the measurements are in the attachment.

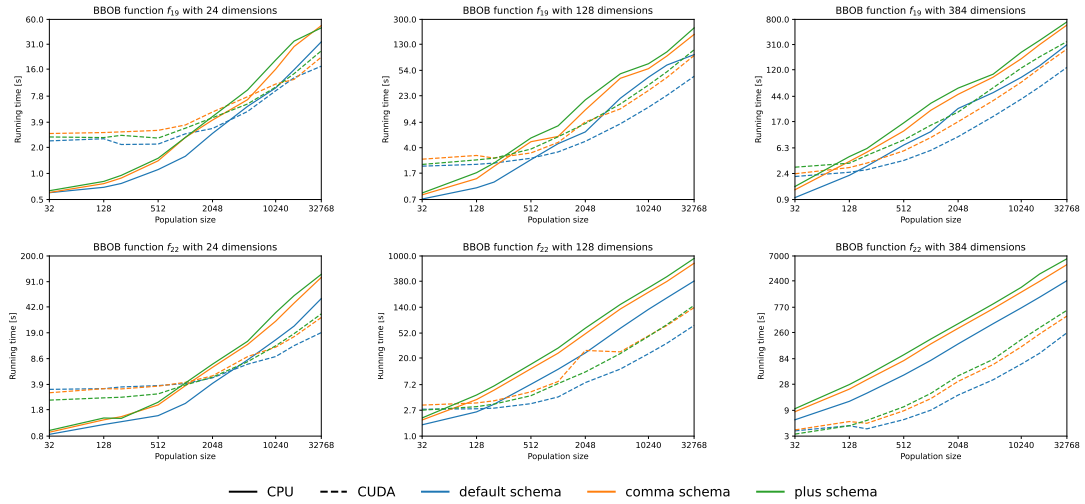


Figure C.12: Running time of real–coded evolutionary algorithm with various crossover schemes. I measure the algorithm on BBOB functions f_{19} and f_{22} with 24, 128, and 384 dimensions. Algorithm used hyperparameters specified in Table B.4 and run for 1000 generations.

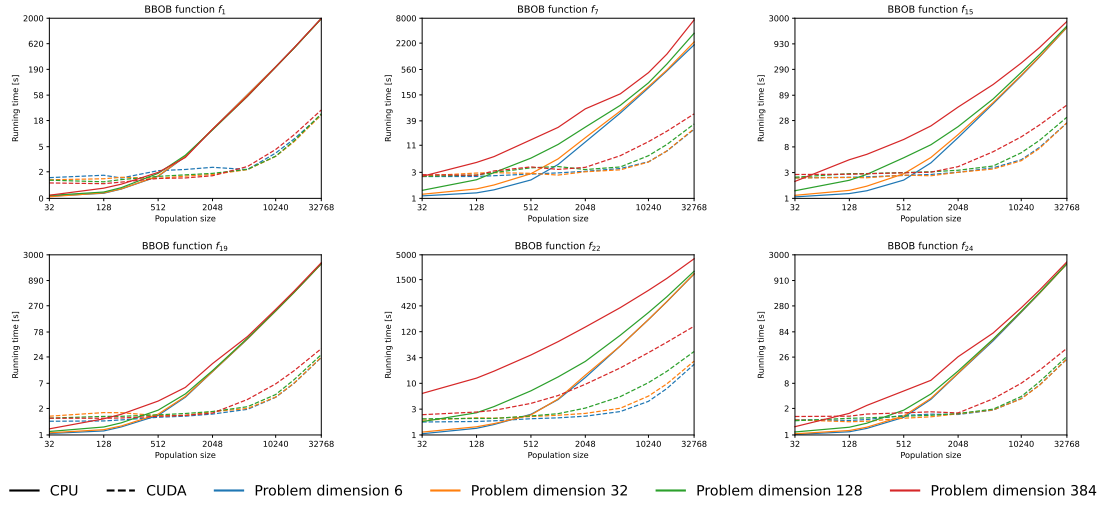


Figure C.13: Running time of Standard Particle Swarm Optimization 2006 algorithm using problems of dimension 6, 32, 128, and 384. The algorithm run for 1000 generations. Populations with over 1000 particles takes advantage of GPU and are clearly faster to evaluate.

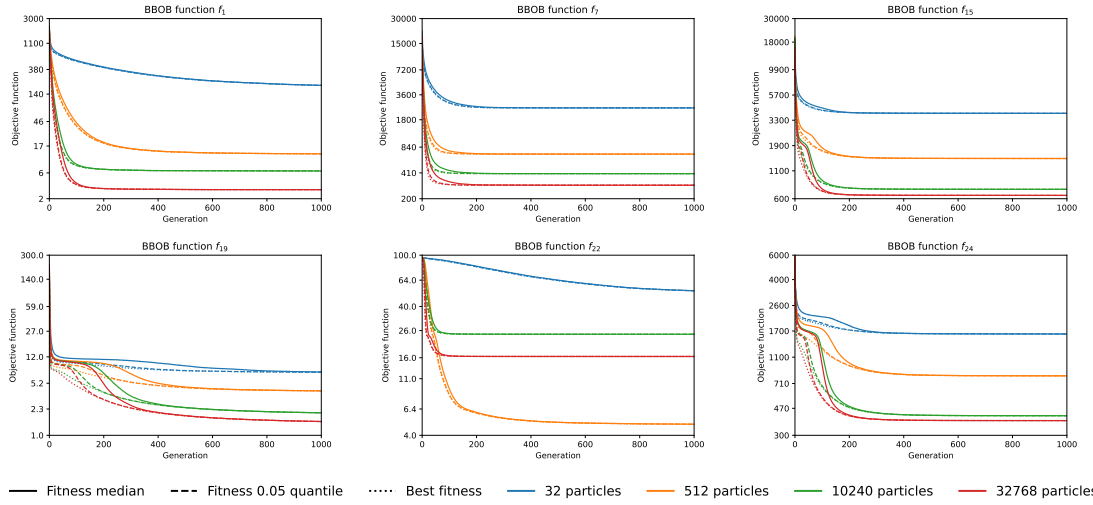


Figure C.14: Median, 0.05 quantile, and best fitness of Standard Particle Swarm Optimization 2006 algorithm using random neighborhood on problem with 128 dimensions. I measured populations consisting of 32, 512, 10240, and 32768 particles. All the problem functions take advantage of more particles, except of function f_7.

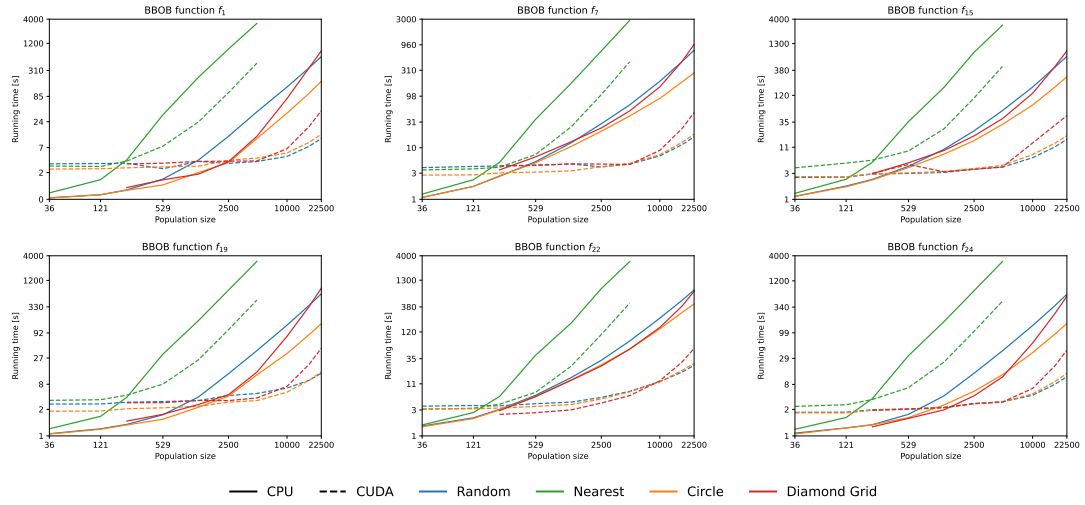


Figure C.15: Running time of Standard Particle Swarm Optimization 2006 algorithm with different neighborhood types. The algorithm run for 1000 generations on problem with 128 dimensions. Neighborhood sizes are at Table B.5. The nearest neighborhood topology was run only to population of size 4900, as it compares all pairs of individuals and the device did not have enough memory. The minimum population size for grid topologies were 225, because topologies size were specified as a fraction of population size and for smaller population neighborhood could not be constructed. The grid topologies were always assembled into square. Running times of circle, linear grid, compact grid, and diamond grid were almost identical (depending only on the size of neighborhood, see Chapter 4), so I plot only measurements for circle and diamond grid.

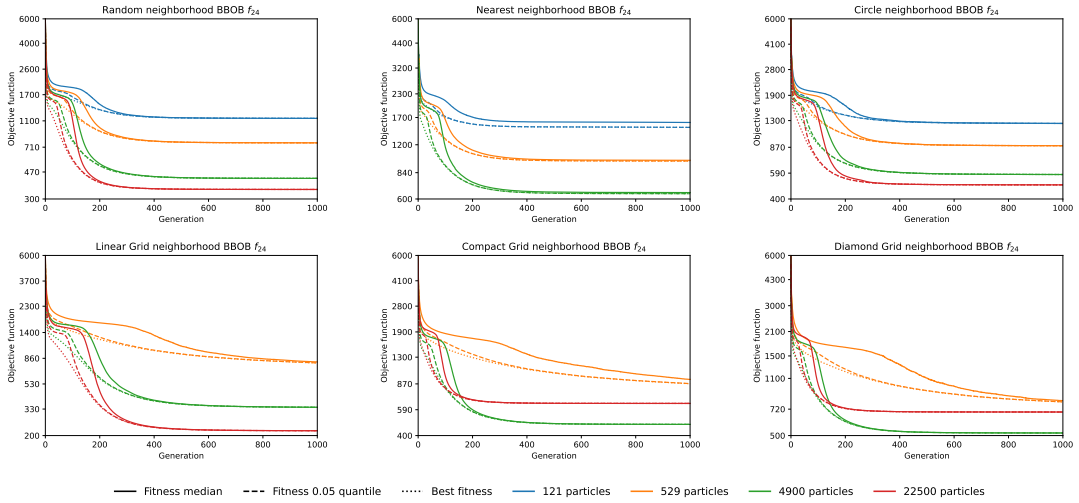


Figure C.16: Fitness of neighborhoods using 121, 529, 4900, and 22500 particles and SPSO–2006 update algorithm. Nearest neighborhood for 22500 particles is not present, because of the memory demands. Grid neighborhoods for 121 particles is not present, because there was not enough particles to build it. All the neighborhoods clearly benefit from greater number of particles with the exception of compact grid and diamond grid. This is probably caused by premature convergence rather than degradation of the performance. Measurements for BBOB functions f_1, f_7, f_{15}, f_{19}, and f_{22} report similar properties and these are in the attachment.

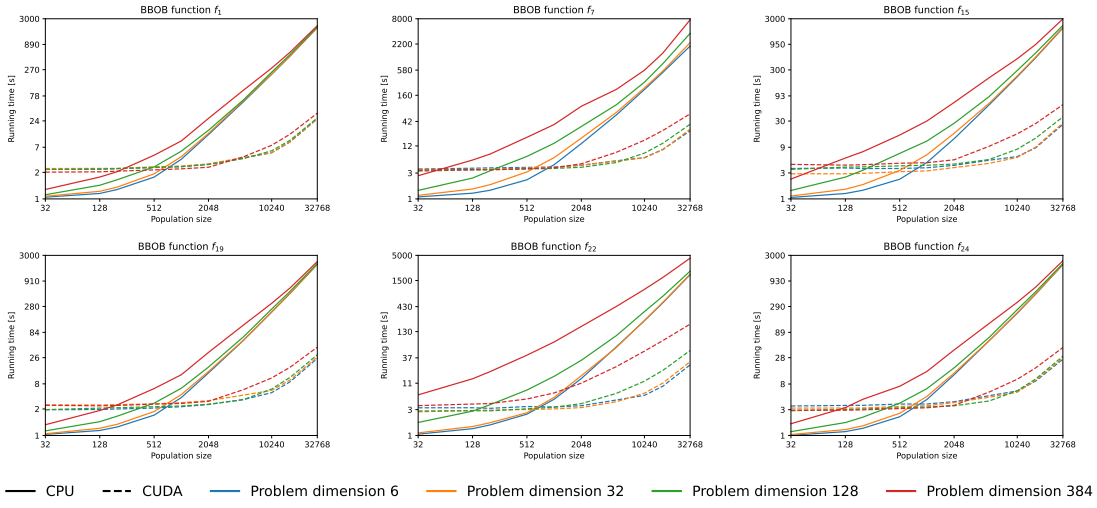


Figure C.17: Running time of Standard Particle Swarm Optimization 2011 algorithm using problems of dimension 6, 32, 128, and 384. The algorithm run for 1000 generations. Populations with over 1000 particles takes clear advantage of GPU .

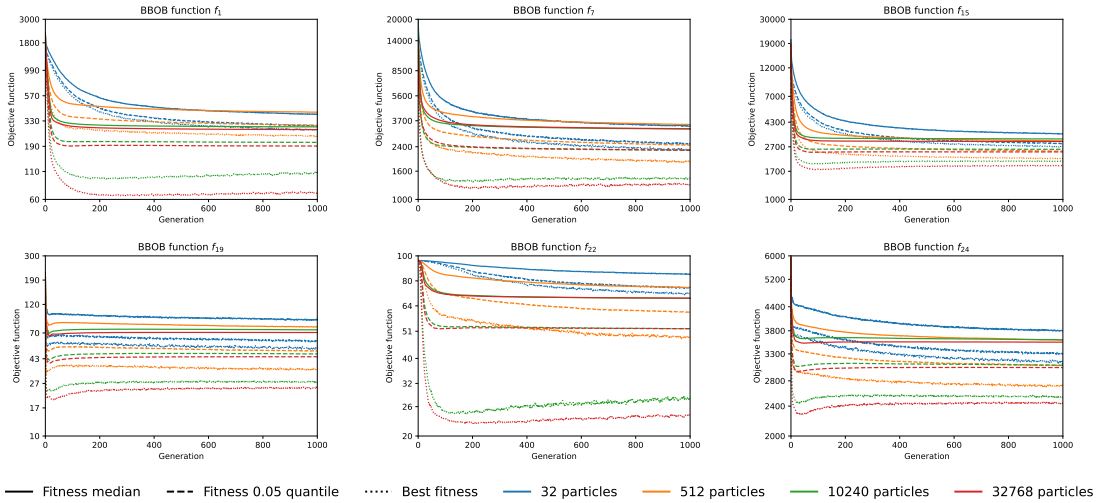


Figure C.18: Median, 0.05 quantile, and best fitness of Standard Particle Swarm Optimization 2011 algorithm using random neighborhood on problem with 128 dimensions. I measured populations consisting of 32, 512, 10240, and 32768 particles. All the problem functions take advantage of more particles.

D. Digital Attachments

Digital version of this work with all the attachments is also available at my GitHub repository on `https://github.com/PatrikValkovic/MasterThesis`.

- `src` — source codes used for this work
  - `BBOBtorch` — implementation of BBOB functions in PyTorch
  - `FFEAT` — implementation of FFEAT library
  - `Scripts` — scripts used for evaluation and measurement
  - `Examples` — some examples of how to use the FFEAT library
- `thesis` — thesis in the LaTeXformat
  - `img` — figures used in this thesis
  - `measurements` — measurements in higher resolution
- `thesis.pdf` — digital version of this thesis
- `README.md` — brief description of the content