Non-Convex Optimization Using GPU

Harshita Kukreja **Hemant Ramawat** **Nikita Anand** **Pranav Jain**

Abstract

Optimization techniques are used in a wide range of applications. Popularly used algorithms Gradient Descent and Genetic Algorithm work well with convex functions but in the case of non-convex ones, they find the local minimum rather than the global. In this work, we make use of GPUs to accelerate the two convex optimization techniques for non-convex problems such that all local optimums can be computed and thus giving us a way to find the overall global optimum. The algorithms are first implemented sequentially to run on the CPU and then compared with the parallelized GPU (three models) version by speedup. The correctness of the program is verified using multiple non-convex functions with known global minimum coordinates. After experimentation, it is observed that both gradient descent and genetic algorithm enjoy a considerable speedup on all three models of GPUs as compared to the sequential version for a reasonable problem size. The code is available on Github.[1]

1 Introduction

As humans, we are faced with optimization problems in our daily life. What is the shortest path to drive to a restaurant? Or is it quicker to take the subway? How do we allocate our time to complete multiple projects at college to meet deadlines?

Optimization lies in the crux of designing several real-world applications used by millions daily. For example, GPS navigation systems find the optimal path from a starting point to a destination taking in consideration constraints like distance, traffic, tolls etc. Other applications are in designing airline reservation systems, resource management in case of natural disasters and financial modeling. It is also a significant part of research in the scientific community; the protein folding problem in biology, parameter training in machine learning, and the idea of minimum power dissipation in physics.

There are several techniques used to solve these various problems and it is worthwhile to make them computationally fast and efficient to aid research. In this work, we concentrate on two of these techniques: Gradient Descent and Genetic Algorithm.

Gradient Descent (Ruder, 2016a) is an optimization technique used widely in many areas like machine learning, statistics, control engineering, mechanical engineering, and game design. Its most prevalent form is arguably in the world of machine learning where it is used to minimize a cost function and train models. It is an iterative algorithm that finds the local maxima/minima of the function it's working on. Gradient Descent can work on functions that are differentiable and convex. This means that it works to find the local optimum in a function and can not find the global in case of non-convex problems.

The second technique (Alam et al., 2020), the genetic algorithm is a heuristic method derived from the evolution process observed in nature. It randomly samples from the domain of the function to optimize to create what is called an initial population. Then it evaluates the fitness of the population using the objective function. It then follows a process to change the population which is described in detail in the following sections. This is done till the iterations are over and the optimum of the final population is computed.

We apply gradient descent and genetic algorithm to nonconvex functions that have multiple local minima. The way we approach this is to randomly sample several starting points and use the convex

[1] https://github.com/hemant2491/GPU-Project

optimization techniques to find all the local minima. The minimum of all local minima then gives us the global minimum of the function. This approach can be parallelized to be faster than the sequential implementation as finding each local minimum is an independent task. We make use of GPUs and their parallel architecture to take advantage of this fact and get improved performance. Our experiments are performed on the NYU cluster using three different models of NVIDIA GPUs as in Table 2.

2 Literature Survey

The disparate literature available for non-convex optimization built on GPUs and multicore processors is vast, so here we summarize very closely related works focusing on gradient descent and genetic algorithm to solve non-convex optimization. There are recurrent examples of solving convex optimization, one of them being Srivastava et al., 2013 using an evolutionary algorithm (EA) based approach to enhance the performance by roughly 160 times while implementing plausibility computations. Ou et al., 2022 presents another example by minimizing near-corner initialization with the use of the genetic algorithm and getting a speedup on GPU of 83 times compared with its sequential counterpart. This forms the basis of non-convex optimization, with one of the common approaches being lowering the complexity bound for global optimization (Nesterov, 2018) which helps find the global minimizer. Problems like Phase Retrieval, which already have a convergence result, can also be solved using non-convex optimization by making simple assumptions (Vaswani, 2020).

Gradient descent optimization algorithms and their challenges are analyzed in Ruder, 2016b. Maneuvering non-convex optimization with the help of gradient descent, Lei et al., 2020 demonstrates a stable foundation for stochastic gradient descent (SGD). It removes the boundedness assumption with no effect on the convergence rate. Huo and Huang, 2016 achieves similar results by using variance reduction to converge the problem instead. Considering a distributed approach for optimizing the problem using SGD, Yu et al., 2019 proves the linear speedup property and reduces performance bottleneck. Another asynchronous distributed and lock-free parallel implementation of SGD by Mohamad et al., 2022 leads to comparable speedup with respect to the number of cores against the state-of-the-art algorithms.

On the other hand, Genetic Algorithm (GA) Alam et al., 2020, derived from natural selection, has started being a reliable adaptive technique for solving complex problems as well. As part of evolutionary algorithms, GA comes under the category of heuristics and is used in emerging areas for hybridization. Non-convex optimization is one such problem that utilizes GA and parallelization in CUDA architecture. Khalily-Dermany and Darabpour, 2018 provides a comparison between different implementations using EAs in centralized and parallelized approaches on CUDA. Nonconvex Mixed Integer Programming Khalily-Dermany and Darabpour, 2018 considers a stochastic algorithm built on an adaptive genetic algorithm. The authors improve performance by hybridization with adaptive resolution local search operator and obtain a speedup of up to 20 times for difficult problems. They also study the various aspects of parallelizing the simulations on the CUDA framework.

3 Problem Definition

3.1 Gradient Descent

First, let us consider how gradient descent works. Suppose the objective function to optimize is $f(x_1, x_2...x_n)$. The values of x_i where $i \in 1, 2..n$ are initialised randomly. Then the following update is performed in an iterative manner until convergence-

$$x_i := x_i - \eta \frac{\partial}{\partial x_i} f(x)$$

This update is carried out simultaneously for all x_i. The term $\frac{\partial}{\partial x_i} f(x)$ is the slope or gradient of the function corresponding to x_i. The equation above essentially repeatedly takes a step in the direction where the slope decreases the most. The symbol η refers to the learning rate which is a parameter that determines how fast the algorithm converges. If the learning rate is too large, the algorithm may skip the minimum and never converge. On the other hand, if it is too small, it will be too slow thus it is important to pick one that is right for the problem.

The step size or the amount by which the values are updated is determined by the learning rate. The algorithm converges when the step size approaches

zero. This is how the minimum of a convex function is reached or a local minima in the non-convex case.

3.2 Genetic Algorithm

As described by Alam et al., 2020, the genetic algorithm is a meta-heuristic motivated by the evolution process and belongs to the large class of evolutionary algorithms in informatics and computational mathematics. These algorithms are frequently used to optimize any class of objective functions.

A genetic algorithm could be summarized using the following steps:

1. Randomly compute the initial population P.
2. Obtain the fitness of the population.
3. Repeat from Step 4 to 7 until convergence.
4. Choose any parent from the population individually.
5. Generates a new population through the crossover process.
6. Insert random genes in a new population to perform mutation.
7. Obtain fitness for newly generated populations.

The population is defined as a set of points lying in the domain of the objective. The fitness of a population is the value of the objective function evaluated at the population.

Suppose each point is d dimensional. In our implementation of crossover, two points are selected randomly and are called the parents. The offspring is computed by copying the first $d/2$ dimensions of the first parent to the first $d/2$ dimensions of the offspring and the next $d/2$ dimensions are copied from the other parent.

For mutation, we randomly multiply the offspring by a value between $[-2, 2]$.

We repeat this process for a fixed number of iterations (fixed to 1000) and compute the minimum of the final population.

4 Proposed Idea

To minimize a non-convex function, we try to use a convex minimization scheme and apply it to a bunch of randomly sampled points. In our work, we utilized gradient descent and genetic algorithm as convex solvers.

Once we randomly sample data points from the domain, we allow the convex solver to minimize the given objective function starting from each of the randomly sampled points. Since, for each data point, the minimization process is independent of others, we utilize GPUs and let each thread work on every data point. Finally, once every thread has minimized the given non-convex function, we compute the minimum of all the solutions computed by each thread and treat the minimum as the global minimum. Algorithm 1 summarizes this method.

4.1 GPU Optimizations

We require an objective function to minimize and a convex solver as input. Next, we randomly sample data points that act as the starting point. In our work, we uniformly sample points from the domain. We sample uniformly to ensure that we have at least one starting point for any region in the domain. This way we can guarantee at least one thread to reach the global minimum.

To launch the kernel, we divide the randomly sampled points into multiple bins and each block is scheduled to work on a single bin. For example, if we have N randomly sampled points, and if we divide it into k bins, the grid dimension or number of blocks will be k and the number of threads per block will be N/k. This way of launching the kernel allows the bin of initial starting points to be stored in the shared memory (tiling) which results in faster access and hence better performance. Since we use iterative convex methods, the thread needs to read and update its starting point in every iteration. Storing it in shared memory allows faster read/write operations as compared to storing it in the global memory.

Also, since each block is given a continuous subset of the global array of initial random points, the memory coalesces which further gives a performance boost.

Once every thread has finished working, we

compute the minimum of all the candidates. The minimum is computed by the first thread ($threadIdx.x = 0$) of every block. Therefore, if we had k blocks scheduled, we would have k minimums computed. To get the global minimum, we again launch the same kernel with only one block and k threads. Thus we perform the optimization again using the minimums earlier computed as the initial points. This results in an accurate solution. Since we have only one block, there would be a single minimum computed which we treat as the global minimum.

For better performance, we also tried a few other optimizations like loop unrolling and minimizing branch divergence.

Algorithm 1 Algorithm to minimize non-convex function

Input: Function F to minimize
$X \leftarrow$ set of randomly sampled points
$M \leftarrow []$
for $x \in X$ **do**
  minimize F starting from x
  let m be the minimum computed
  $M \leftarrow m$
end for
return $min(M)$

5 Experiments

5.1 Correctness

To show the correctness of our method, we test the method on a bunch of non-convex objective functions. The following sections mention the objective function used, its domain, global minima, and the minimum computed by our model.

5.1.1 Simple Parabolic Function

$$F(x) = \sum_{i=1}^{2} (x_i - 5)^2$$

Domain: $x_i \in [-10.0, 10.0]$
Global minimum: $f(x^*) = 0, x^* = (5, 5)$

Gradient Descent (CPU):
$f(x^*) = 1.22\text{e}{-6}, x^* = (4.999915, 4.998896)$
Gradient Descent (GPU):
$f(x^*) = 4.94\text{e}{-7}, x^* = (5.000224, 4.999334)$
Genetic Algorithm (CPU):
$f(x^*) = 2.63\text{e}{-5}, x^* = (5.000525, 4.994891)$

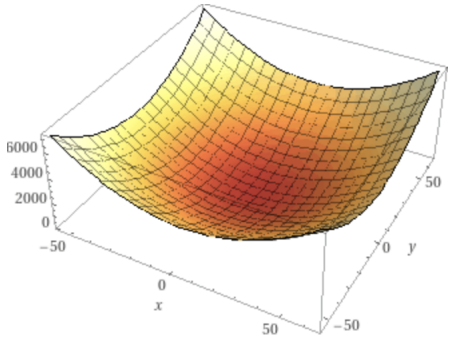

Figure 1: Plot of simple parabolic function

Genetic Algorithm (GPU):
$f(x^*) = 3.63\text{e}{-6}, x^* = (4.998106, 5.000212)$

5.1.2 Polynomial Curve with 2 Minima

$$F(x) = x^4 - 2x^2 + 3$$

Domain: $x_i \in [-10.0, 10.0]$
Global minimum: $f(x^*) = 2, x^* = \{-1, 1\}$

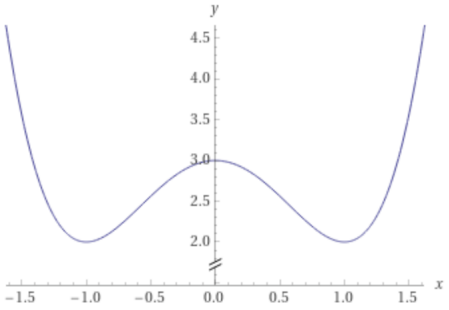

Figure 2: Plot of the polynomial curve with 2 minima

Gradient Descent (CPU):
$f(x^*) = 2.00, x^* = -1.00$
Gradient Descent (GPU):
$f(x^*) = 2.00, x^* = 1.00$
Genetic Algorithm (CPU):
$f(x^*) = 2.00, x^* = 1.00$
Genetic Algorithm (GPU):
$f(x^*) = 2.00, x^* = -1.00$

5.1.3 Rastrigin Function

$$F(x) = 20 + \sum_{i=1}^{2} (x_i^2 - 10\cos(2\pi x_i))$$

Domain: $x_i \in [-5.12, 5.12]$
Global minimum: $f(x^*) = 0, x^* = (0, 0)$

Gradient Descent (CPU):
$f(x^*) = 1.23\text{e}{-10}, x^* = (-7.86\text{e}{-7}, 4.52\text{e}{-8})$
Gradient Descent (GPU):

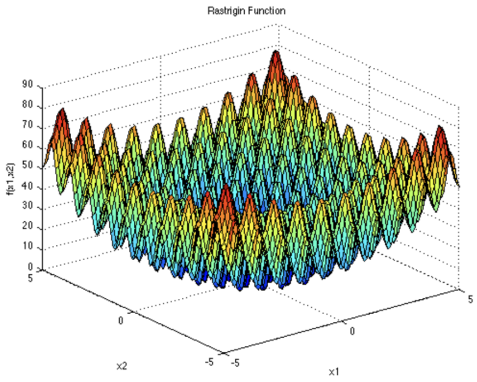

Figure 3: Plot of Rastrigin function

$f(x^*) = 1.21\text{e}{-10}, x^* = (5.94\text{e}{-7}, -5.11\text{e}{-7})$
Genetic Algorithm (CPU):
$f(x^*) = 0.00, x^* = (3.19\text{e}{-31}, -5.65\text{e}{-39})$
Genetic Algorithm (GPU):
$f(x^*) = 0.00, x^* = (-2.78\text{e}{-10}, 1.71\text{e}{-10})$

We can see that our models converge to the global minima proving the correctness of our implementation.

In the next section, we provide experiments and analysis in terms of performance. For all the timings stated, we use the parabolic function as the objective function.

6 Analysis

6.1 Experimental set-up

To compare the speed-up between the sequential version and the parallelized version of the Genetic Algorithm and the Gradient Descent algorithm we have used below values of the parameters:

Parameters	***Values***
Grid Dimensions / No. of Chromosomes	1*1 / 2*2 / 4*4 / 8*8 / 16*16 / 32*32 / 64*64 / 128* 128 / 256*256

Table 1: Different values of parameters for experiments

For our experiments we used three models of NVIDIA GPUs available on the NYU CIMS cluster with specifications stated in Table 2.

6.2 Parallelization Gains

Upon executing the same algorithms sequentially over CPU and then across multiple GPUs, as expected, when the number of data points in the optimization problem are increased the sequential execution times rose exponentially and we saw speed-ups upto 30 times. The optimization problem algorithms benefited from the parallelization. The parallelization implementation of the algorithms is also better for better and newer GPUs.

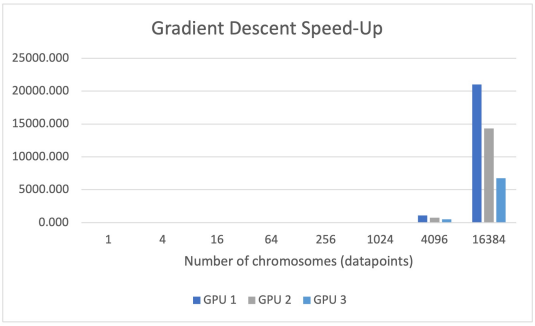

Figure 4: Plot of speedup for gradient descent

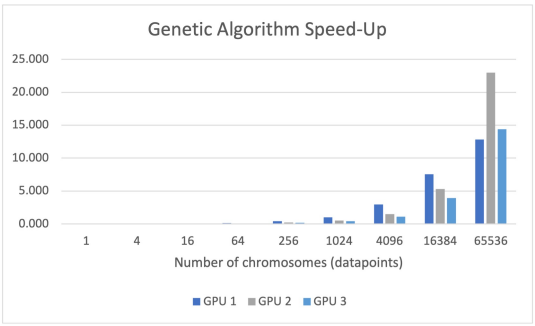

Figure 5: Plot of speedup for genetic algorithm

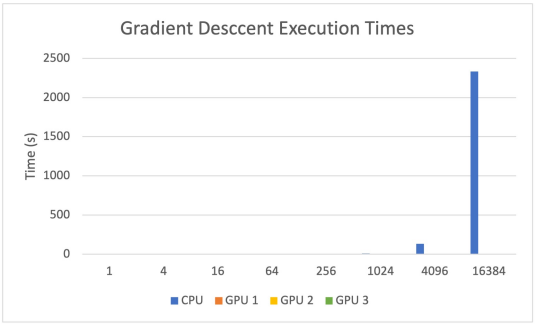

Figure 6: Plot of execution times for gradient descent on different GPUs

7 Challenges Faced

Gradient descent is an inherently sequential algorithm and there is only so much parallelization that can be done at any level. We cannot have a couple of threads computing the gradient descent on a single point which was one of the challenges faced in the work. The genetic algorithm requires sorting

	GPU-1	GPU-2	GPU-3
Manufacturer	NVIDIA	NVIDIA	NVIDIA
Model	GeForce GTX TITAN	GeForce RTX 2080 Ti	TITAN V
Architecture	Kepler	Turing	Volta
Cores	2880	4352	5120
Max Clock Rate	0.98 GHz	1.63 GHz	1.46 GHz

Table 2: Hardware Specifications for the GPUs used

Genetic Algorithm									
Chromosomes	**Rastrigin**			**Polynomial Curve**			**Parabolic**		
	CPU	GPU-2	Speedup	CPU	GPU-2	Speedup	CPU	GPU-2	Speedup
1	0.006	0.168	0.036	0.006	0.195	0.031	0.005	0.256	0.020
4	0.008	0.188	0.043	0.014	0.185	0.076	0.006	0.234	0.026
16	0.012	0.190	0.063	0.042	0.188	0.223	0.007	0.245	0.029
64	0.030	0.191	0.157	0.186	0.186	0.844	0.015	0.182	0.082
256	0.100	0.193	0.518	0.705	0.187	3.548	0.05	0.241	0.207
1024	0.303	0.190	1.595	3.430	0.187	18.465	0.126	0.236	0.534
4096	1.200	0.198	6.061	18.390	0.194	94.794	0.357	0.239	1.494
16384	5.380	0.233	23.090	72.265	0.225	321.178	1.336	0.252	5.302
65536	24.781	0.428	57.899	355.903	0.363	980.449	6.207	0.27	22.989

Table 3: Timings computed for Genetic Algorithm

Gradient Descent									
Starting Points	**Rastrigin**			**Polynomial Curve**			**Parabolic**		
	CPU	GPU-2	Speedup	CPU	GPU-2	Speedup	CPU	GPU-2	Speedup
1	0.012	0.215	0.056	0.008	0.367	0.022	0.005	0.242	0.021
4	0.005	0.169	0.030	0.033	0.323	0.102	0.006	0.152	0.039
16	0.007	0.201	0.035	0.287	0.371	0.774	0.012	0.211	0.057
64	0.021	0.231	0.091	0.009	0.366	0.025	0.078	0.21	0.371
256	0.239	0.222	1.077	0.131	0.39	0.336	0.569	0.229	2.485
1024	2.142	0.193	11.098	1.142	0.563	2.028	7.971	0.217	36.733
4096	39.613	0.138	287.051	6.237	0.344	18.131	131.873	0.175	753.560
16384	552.849	0.149	3710.396	44.971	0.629	71.496	2.33e3	0.163	1.43e4
65536	8.34e3	0.174	4.7e4	402.739	1.722	233.879	4.42e4	0.203	2.18e5

Table 4: Timings computed for Gradient Descent

for the selection process. Since recursion on GPU is discouraged, we used the $O(N^2)$ sorting algorithm for each thread which harms the performance. Also, to find the minimum in each block, we only let the first thread compute the minimum which leads to branch divergence.

8 Conclusions

We were able to successfully extend Gradient Descent as well as the Genetic Algorithm to non-convex optimization problems. The correctness was verified on both simple and complex convex functions with known global minimums.

Both algorithms enjoy favorable speedups on all three models of GPUs over the CPU when the problem size is large enough. Overall that number was found to be 4096 points or chromosomes for both techniques. The performance increases drastically with the increase in problem size. The speedup for the genetic algorithm on the polynomial curve goes as high as 980 times for 65536 points and for gradient descent the greatest speed up of 47000 times was again with the

	Genetic Algorithm			Gradient Descent		
Chromosomes/Points	GPU-1	GPU-2	GPU-3	GPU-1	GPU-2	GPU-3
1	0.123	0.256	0.307	0.115	0.242	0.349
4	0.111	0.234	0.35	0.114	0.152	0.276
16	0.118	0.245	0.341	0.12	0.211	0.284
64	0.119	0.182	0.387	0.116	0.21	0.345
256	0.124	0.241	0.328	0.105	0.229	0.313
1024	0.129	0.236	0.305	0.119	0.217	0.27
4096	0.121	0.239	0.331	0.12	0.175	0.27
16384	0.177	0.252	0.342	0.111	0.163	0.346
65536	0.484	0.27	0.431	0.138	0.203	0.323

Table 5: Execution timings across GPUs for Parabolic function

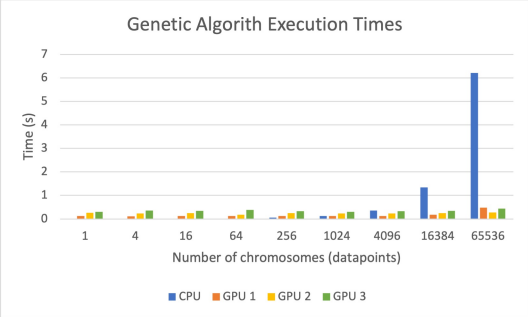

Figure 7: Plot of execution times for genetic algorithm on different GPUs

largest number of starting points for the parabolic function. The improvements in speed can be attributed to the fact that the problem has many similar independent computations that can utilize the parallel architecture in the GPUs well.

Our results indicate that this is a GPU-friendly problem and can be used in its parallelized form for increased performance. This allows one to approach non-convex problems with convex optimization techniques while remaining computationally feasible.

References

Tanweer Alam, Shamimul Qamar, Amit Dixit, and Mohamed Benaida. 2020. Genetic algorithm: Reviews, implementations, and applications. *CoRR*, abs/2007.12673.

Zhouyuan Huo and Heng Huang. 2016. Asynchronous stochastic gradient descent with variance reduction for non-convex optimization.

Mohammad Khalily-Dermany and Roya Darabpour. 2018. A comparison between evolutionary-algorithms parallelization in CUDA architecture. In *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. IEEE.

Yunwen Lei, Ting Hu, Guiying Li, and Ke Tang. 2020. Stochastic gradient descent for nonconvex learning without bounded gradient assumptions. *IEEE Transactions on Neural Networks and Learning Systems*, 31(10):4394–4400.

Saad Mohamad, Hamad Alamri, and Abdelhamid Bouchachia. 2022. Scaling up stochastic gradient descent for non-convex optimisation. *Machine Learning*, 111(11):4039–4079.

Yurii Nesterov. 2018. *Lectures on Convex Optimization*. Springer International Publishing.

Junlin Ou, Seong Hyeon Hong, Paul Ziehl, and Yi Wang. 2022. GPU-based global path planning using genetic algorithm with near corner initialization. *Journal of Intelligent &amp Robotic Systems*, 104(2).

Sebastian Ruder. 2016a. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

Sebastian Ruder. 2016b. An overview of gradient descent optimization algorithms.

Rupesh Kumar Srivastava, Kalyanmoy Deb, and Rupesh Tulshyan. 2013. An evolutionary algorithm based approach to design optimization using evidence theory. *Journal of Mechanical Design*, 135(8).

Namrata Vaswani. 2020. Non-convex structured phase retrieval.

Hao Yu, Rong Jin, and Sen Yang. 2019. On the linear speedup analysis of communication efficient momentum SGD for distributed non-convex optimization. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7184–7193. PMLR.