GPU and Multi-Processing Implementation of a Swarm Intelligence Algorithm

Jubril-Awwal Shomoye

April 2022

Acronyms

ACO Ant Colony Optimisation. 6, 9

AGPM All-GPU Parallel Model. 18

AMP Asymmetric Multi-Processing. 13

ARM Association Rule Mining. 16

BSO Bee Swarm Optimisation. 16

CPU Central Processing Unit. 6

CS Cuckoo Search. 10

CUDA Compute Unified Device Architecture. 12

DFO Dispersive Fly Optimisation. 6, 13, 17

FASI Framework for Accelerating Swarm Intelligence. 16

FPGA Field Programmable Gate Arrays. 16

FWA Firework Algorithm. 14

GA Genetic Algorithm. 6, 7

GMEL GPU-based MEtaheuristics Library. 17

GPGPU General Purpose GPU. 11

GPU Graphical Processing Unit. 6

HJDS Hooke-Jeeves Direct Search. 15

MIM Multiswarm Island Model. 19

MP Multi-Processing. 6, 13

MPM Multiphase Parallel Model. 18, 24

MT Multi-Threading. 13

NPM Naive Parallel Model. 17, 18, 23

PSO Particle Swarm Intelligence. 6, 8

SI Swarm Intelligence. 6, 9, 13

SIA Swarm Intelligence Algorithms. 6, 17, 24

SIB-EA Swarm Intelligence Based Evolutionary. 17

SMP Symmetric Multi-Processing. 13

TSP Travelling Salesman Problem. 9

Contents

Abstract

"I see Moore's Law dying here in the next decade or so" was quoted by Moore (2015) himself with an interview with the IEEE Spectrum. As of recent, There has been a constant plateauing in CPU performance. However, GPUs can contribute in increasing performance. Swarm Intelligence (SI) and nature-inspired computation techniques are known to be a composition of parallel computing. Where multiple processes are being simultaneously executed rather than the conventional normal computation where its done in a serial manner. To further increase the speed of processing. This can be executed through the implementation of a Graphical Processing Unit (GPU) . Due to GPUs contributing largely to parallelism, solutions can be made faster when compared to a non GPU-based swarm intelligence computation. Another implementation could be Multi-Processing (MP), where multiple processes are executed simultaneously by more than one computer processor. Such features will be applied to the relatively newly introduced swarm intelligence algorithm. Dispersive Flies Optimisation.

1 Introduction

Moore's law states that the number of transistors on a microchip, doubles every two years whilst halving the cost of computers. This means that computer would have twice the speed when compared to their previous last two years. However, experts are under the impression that CPUs would hit their physical limit around the 2020s Rotman (2020). Moore himself also believes that this Moore's Law would most likely end in this decade or the next Moore (2015). From Figure 1, it back these assumptions as around 2016 to 2018 CPUs begin to plateau. This shows that CPUs are starting to see a decline in evolution and would most likely see their limit very soon. Although CPUs would see a decrease in performance, computer systems could still retain a good performance with a GPU

1.1 Swarm Intelligence

Instead of reaching a single solution conventionally, Swarm Intelligence (SI) (which was introduced by Beni & Wang (1993)) are population-based algorithms which use a collective number of agents(or objects), whilst continuously communicating with each other to acquire an optimised solution to a problem. Swarm Intelligence Algorithms (SIA) tend to be nature inspired and mimic their inspirations social behaviour to reach a goal. There are many population-based algorithms such as Genetic Algorithm (GA), Particle Swarm Intelligence (PSO), Ant Colony Optimisation (ACO), Dispersive Fly Optimisation (DFO) and many more.

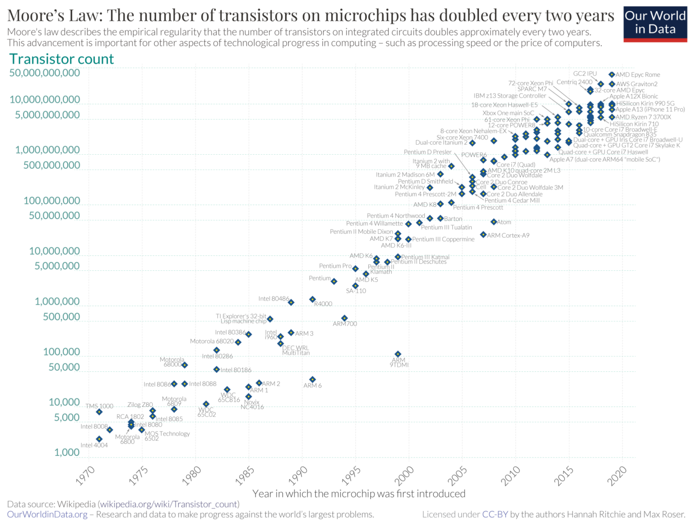

Figure 1: Transistors Count over time Roser & Ritchie (2013)

1.1.1 Genetic Algorithm

”Genetic Algorithm is an evolutionary computation paradigm inspired by biological evolution” Sawai & Kizu (1998). Introduced by Holland (1992), Genetic algorithms (GA) form a simulation where ”agents” are placed into an environment, with limited resources. The agents struggle to survive and the agents that survive longer are rewarded with the ability to pass down their genetic material to the new generation of agents. This continuously occurs until the solution is found. It follows the rules of ”Survival of the fittest” by Darwin (1859)
How it works:

First Population
At the start of the algorithm, a population of N agents are created. Each agent fitted with randomly generated ”genes”. When compared to the real world, the gene would would be the genotype of the agent and whilst its phenotype would be its physical attributes.

Fitness & Selection
To ensure the genetic algorithm is executed properly. The agents of the population would need to be evaluated based of a fitness function. A fitness function is a function which gives an agent, in the population, a numerical value and evaluates whether the agent is close or good to achieving the solution.

Based of the fitness function, the agents with the best fitness value are chosen to provide child and pass down their genetic material to the next generation. This process could be executed either through an elitist method, where the top percentage of the population, e.g. 40%, would be chosen to produce children. The other approach could be fitness proportionate selection (also know as Roulette Wheel Selection) Fogel (2006), where an agent is selected for reproduction in proportion to its fitness value.

Reproduction & Mutation
Now that the agents have been selected, they are ready to be modified. There are two main methods of modifying the agents selected. These are:

- Mutation
- Crossover(recombination)

Mutation changes one agent at a time for modification. Once this agent has been chosen, one element in its genes is chosen to be mutated. For example:

$$[1110] \text{ represents } x_1^t \tag{1}$$

an element would be chosen for mutation which will become,

$$[1111] \text{ which now represents } x_1^{t+1} \tag{2}$$

A child will now be born with the mutated gene [1111].

Unlike mutation, crossover is not done individually but rather done over a large number of solutions. It would take the genes of two parents and combine its genes to form a child.

After some time the parents will die and allow the new generation of children, mutated or not, to continue living and have the opportunist to repeat the cycle once more.

1.1.2 Particle Swarm Optimisation

A population-based stochastic optimisation technique that was inspired by the social behaviour of bird flocking, was introduced by Kennedy & Eberhart (1995). In Particle Swarm Intelligence (PSO), the agents would circle around an environment, in order to reach the solution. After every iteration, a group of agents that are the closest to the best agent of the population will be adjusted. This would continue until one of the agents have reached the solution. At the start of the algorithm, P particle's position is denoted as i at iteration t as $X^i(t)$ with its coordinates in the dimension as

$$X^i(t) = (x^i(t), y^i(t)) \tag{3}$$

with a velocity of each particle as

$$V^i(t) = (v_x^i(t), v_y^i(t)) \tag{4}$$

After each iteration, the position of the particle would be updated and it would be incremented by 1. For example,

$$X^i(t+1) = X^i(t) + V^i(t+1) \tag{5}$$

which could also be read as

$$x^i(t+1) = x^i(t) + v_x^i(t+1) \tag{6}$$

$$y^i(t+1) = y^i(t) + v_y^i(t+1) \tag{7}$$

Simultaneously, the velocities are also updated by the equation

$$V^i(t+1) = wV^i(t) + c_1r_1(pbest^i - X^i(t)) + c_2r_2(gbest - X^i(t)) \tag{8}$$

where r_1 and r_2 are randomly generated numbers between 0 and 1 ($r \sim U(0,1)$). $pbest$[1] is the best position of where particle i had explored whilst $gbest$ is the position which was explored by all particles of the swarm. They are both updated after every iteration to reveal the best position in the dimension. w is defined as the inertia weight constant as is determines the particle's new velocity based of its previous velocity between 0 and 1.c_1 and c_2 are parameters which could be called the cognitive and social coefficient. Both handle the weight of exploring and exploiting in the swarm.

1.1.3 Ant Colony Optimisation

Ant Colony Optimisation (ACO) by Dorigo & Di Caro (1999), was inspired by the way a group of ants communicated with each other to find food. Instead of standard communication like other animals, ants communicate through pheromones. The same method is applied in ACO, where the "ant" agent randomly walks through a completed connected graph in an attempt to find the solution. This type of SI is normally applied to the Travelling Salesman Problem (TSP), where the most optimum path is found.

At the start of the algorithm, a random amount of pheromones, are distributed along the edges of a map. An ant is placed at a random position/node on the map. The ant moves to another edge on the map using the transition rule, where a random map is given a biased pheromone level on each edge. With heuristic information being stored on the edges too. The transition rule is

$$p_k(r,s) = \frac{T(r,s)^\alpha \times H(r,s)^\beta}{\sum_C T(r,c)^\alpha \times H(r,c)^\beta} \tag{9}$$

$p_k(r,s)$ is the probability that ant k moves from node r to s. $T(r,s)$ is the amount of pheromones connecting edge r to s. Making α the strength of the pheromone. $H(r,s)$ is the heuristic value of the edge. The amount of heuristic value depends on the length/distance of the edge. So $H(r,s)$ would be $\frac{1}{distance(r,s)}$. β is denotes the strength off the heuristic information. Finally, c is the amount of cities which have not been visited.

The ant is be unable to return to its previous node when searching, so it will venture round the map, only to return to its original positions when there are no more paths to take. The ant will keep a memory of this path for future ants. For example, the ant's tour memory would be A, E, D, B, C, A, which will now become its fitness value. Since the tour of the ant is completed, the pheromones will undergo a pheromone update with the formula,

$$T(r,s) = \rho \times T(r,s) + \sum_{k=1}^{m} A_k(r,s) \tag{10}$$

Where $A_k(r,s) = \frac{1}{L_k}$. ρ is the decay rate of pheromones. $A_k(r,s)$ is the increment of pheromones added onto edge connected to nodes r and s, via ant k. Lastly L_k is the path taken by ant k. More ant would take the tour memory and increase the pheromone on the tour, whilst pheromones outside the path decrease over time

1.1.4 Cuckoo Search

Another meta-heuristic algorithm, which was developed by Yang & Deb (2009), is Cuckoo Search (CS). It was inspired by the behaviour of some cuckoo species, where they had portrayed some form of parasitism such as laying their eggs in another bird species' nest. In CS, the 'cuckoo' agents are search agents whilst the potential solutions would be the eggs stored in nests. Each egg would serve as new solution. The aim of the entire algorithm would be to replace the current solutions(eggs) with potentially better new solutions.

1.2 Dispersive Flies Optimisation

Al-Rifaie (2014) had introduced a new form of swarm intelligence called Dispersive Flies Optimisation (DFO). It was inspired by the swarming of flies, hovering over a food source. The algorithm consists of two mechanisms; the formation and breaking of the swarm. Each fly is considered an agent and it represents the solution to a problem. The flies, in the population, will communicate with other flies in order to finding the better/best solution. DFO uses ring topology as a form of communication, meaning every fly will have a left and right neighbour.

For example:

$$\leftarrow x_0 - x_1 - x_2 - x_3 \rightarrow \tag{11}$$

Where x is the best fly.

When the DFO algorithm begins its first generation of flies, $t = 0,$ the i^{th} fly's d^{th} dimension are initialised as:

$$x^0_{id} = x_{min,d} + r(x_{max,d} - x_{min,d}) \tag{12}$$

x_{min} & x_{max} are the upper and lower bounds of the d^{th}dimension and $r \sim U(0, 1)$. This ensures that the population is initialised with each fly positioned in a random position in the search space. After every 100^th iteration, each fly updates its self based on its current position, the position of its best neighbour's position and the position of the best fly in the population. Making its update equation become:

$$x^{t+1}_{id} = x^t_{i_nd} + u(x^t_{sd} - x^t_{id}) \tag{13}$$

x^{t+1}_{id} will become the next position of the i^{th} fly in the dimension d. $x^t_{i_nd}$ is the position of x^t_i's best neighbour in dimension d. $u \sim U(0, 1)$ is newly generated after every dimension update. x^t_{sd} is the position of the best fly, in the population, in dimension d. x^t_{id} is the position of the i_{th} fly at the time step t, in the dimension d. Due to the possibilities of flies being disturbed in the swarm, DFO would have a disturbance threshold(Δ) of 0.001. Before updating a fly's dimension if a random number being generated was lower than Δ (e.g. $U(0, 1) < \Delta$), then the dimension of the fly would be restarted.

Mohammad had shown that DFO had outperformed generic population based algorithms such as differential evolution, particle swarm optimisation and genetic algorithm. He also shows that DFO is 84% more efficient and 90% more reliable than the other algorithms

1.3 GPU

Initially, Graphical Processing Units (GPUs) were designed to aid image and graphic processing in computers, as parallel computed was desired to handle such intense computation. Due to modern day high demands, GPUs have been forced to evolve into "a full-fledged parallel programmable processor with additional fixed-function on special-purpose functionality" Owens et al. (2008). A GPU is fitted with more transistors, when compared to a CPU, and these transistors are used on data processing. When compared to a CPU, these transistors speed up the rate of computation and allows the GPU to execute more float point operations per second. Due to its great performance, it has enabled GPUs to be used for general-purpose computation. Which coins the term General Purpose GPU (GPGPU), where GPU programming is enacted. This implementation enables highly intensive computational at a lower run-time. However, for a user to start GPU programming, they must first understand how to use CUDA.

1.3.1 CUDA

Created by NVIDIA (2015), Compute Unified Device Architecture (CUDA) is a parallel computing platform which is integrated into NVIDIA GPUs. It uses C/C++ programming language, allowing programmers to utilise the GPU's parallel features when executing CUDA kernels.

1.3.2 Kernels, Threads, Grids and Blocks

Kernels or CUDA Kernels are parallel functions that handles the transfer of data between the CPU(host) and GPU(device) for the execution of a program. When a CUDA kernel transferred from the CPU to a GPU, it is executed by a group of threads. A group of threads are fitted and sorted in thread blocks. These thread blocks efficiently share data through fast shared memory by cooperating with each other, whilst synchronizing the number of executions to equal memory accesses.

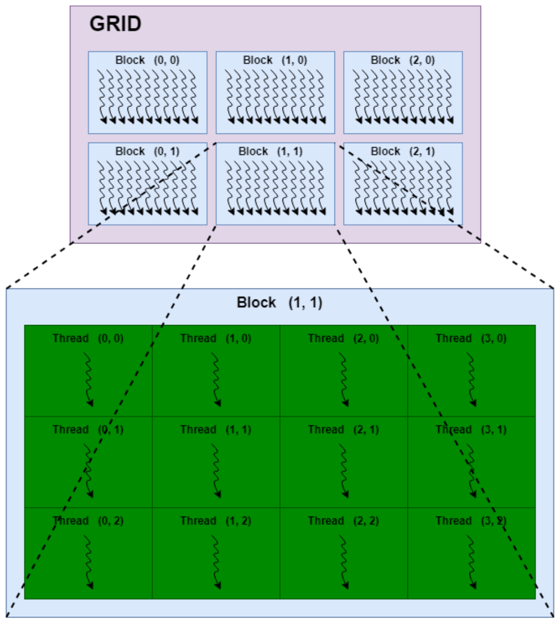

Figure 2: Grid of Thread Blocks

Threads are sorted into single blocks and these group of blocks make up a grid. Figure 2 illustrates this. A program could specify a block as an array, up to 3 dimensions and identifies each thread using up to 3 component indexes. For example, a 2-dimensional block with a size of $D_x \times D_y$, the thread ID of (x, y) would be $y * D_x + x$. CUDA threads could access the data stored in multiple memory spaces as they are executed. Every thread is given a private register and local memory. The thread blocks have a shared memory which is available to all the threads inside the block.

1.4 Multi-Processing

Multi-Processing (MP) is the simultaneous execution of multiple processes via more than one computer process. Such method can increase the computation power, however, this method could be considered time-consuming. This is normally achieved by the CPU and its processors. Multi-Processing can be split into two groups, Symmetric Multi-Processing (SMP) and Asymmetric Multi-Processing (AMP). Symmetric Multi-Processing is parallel processing where all processors are treated equally when solving a problem. They would have access to the same memory and the same data through communication. It does require a lot of power. Whereas, Asymmetric Multi-Processing (AMP) doesn't treat its processors equally. There is a 'master' processors which handle the priority of other processors. Meaning access to memory and data is all dependent on the master processor.

Multi-Processing should not be mistaken by Multi-Threading (MT). Like MP, Multi-Threading many threads are created from an application to increase computational power. However, the process is executed economically and doesn't have any groups it could be split into.

The aim of this project is to present the affect of parallelisation in Swarm Intelligence. This would be shown via a GPU or a multi-processed version of the SI. The Swarm Intelligence Algorithm(SIA) which would be the main focus in this project, will be the Dispersive Fly Optimisation (DFO) algorithm

2 Literature Review

2.1 GPU-based SI

The first instances of GPU-Based Swarm Intelligence was Catala et al. (2007) and Li et al. (2006). Li et al. (2006) had modified the PSO algorithm to run on the GPU, where the particles used the texture-rendering and mapping feature of the GPU. The modification was tested on three benchmark functions. It was reported that there was a 4.3x increase in speed when compared with the original PSO, with the population size at 6400 particle. Catala et al. (2007) used vertex shader and fragment shader of the GPU on the ACO algorithms. They used these features to build the ant's paths and so solve and orienteering problem.

The vertex shader presented a 1.3x increase in speed while the fragment shader conferred a 1.45x speedup.

2.1.1 GPU Firework Algorithm

Ding et al. (2013) performs a study where they measured the performance between a modified Firework Algorithm (FWA), to compare it to the original FWA and PSO(Particle Swarm Optimisation).The modified FWA consisted of the original FWA, however it is has been modified to accommodate the GPU parallel architecture. Figure 1 shows the Framework of the Firework Algorithms. Using benchmark functions to test the performance. It is presented that the GPU-FWA had outperformed the FWA and the PSO. Ding had concluded that the GPU-FWA could be a powerful tool when attempting to solve large-scale optimisation problems.

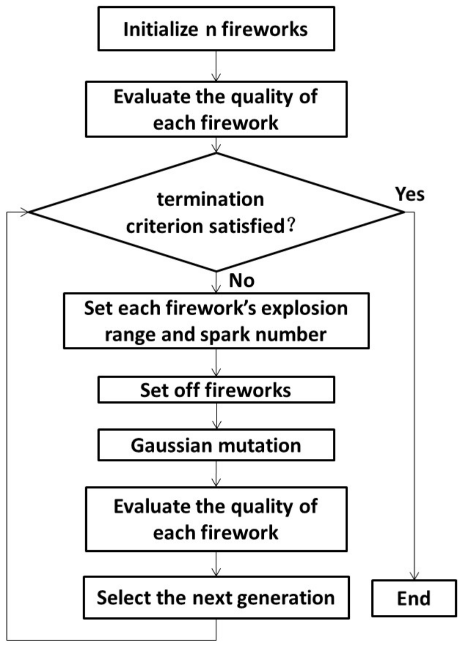
Initialize n fireworks
Evaluate the quality of each firework
termination criterion satisfied?
Yes
No
Set each firework's explosion range and spark number
Set off fireworks
Gaussian mutation
Evaluate the quality of each firework
Select the next generation
End

Figure 3: Framework of Fireworks Algorithm by Ding et al. (2013)

2.1.2 GPU-Based Hybrid jDE Algorithm

Boiani & Parpinelli (2020) had used "a high-performance hybrid algorithm to approach the 3D-AB off-lattice model through Graphic Processing Units(GPUs)". The study showed that through the use of the GPU large parallel architecture, you would be able to ”test larger protein sequences without compromising the running time”. Boiani used the Hooke-Jeeves Direct Search (HJDS) algorithm, which known to be computationally expensive, and suggested that they would find more ”efficient and parallelisable local search algorithms” to replace the HJDS algorithm and run the better algorithm using a GPU as a future solution to their problem.

2.1.3 A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms

Tan & Ding (2015) divided the Swarm Intelligence Algorithms(SIAs), to be implemented into GPU-Based architectures, into 4 categories. The four categories were:

- Naive Parallel Model.
- Multiphase Parallel Model.
- All-GPU Parallel Model.
- Multiswarm Island Model.

When compared to a CPU-Based implementation, GPU-Based Implementation was more superior. However, Tan pointed out that ”although CUDA is the dominate platform for GPU-based implementation of SIAs, it suffers from the drawback of closed environment as only NVIDIAs GPUs were supported.” With this disadvantage with CUDA, OpenCL was suggested to be a better alternative. It is also supported by multiple CPUs and GPUs such as Windows 10 64-bit running on Intel®Pentium®Processor J6426, AMD Radeon (TM) R9 Fury Series and AMD Radeon (TM) R7 M360. 2.4 GPU-Accelerated Data Mining with Swarm Intelligence

2.1.4 AntMinerGPU & ClusterFlockGPU

Weiss (2010) looked at the possibility whether a GPU parallel computing model would increase the accuracy and efficiency of two types of SIA which were used for data mining. Weiss used the algorithms:

- ”AntMinerGPU, an ant colony optimization algorithm for rule-based classification”
- ”ClusterFlockGPU, an ant colony optimization algorithm for rule-based classification”

When compared to the CPU-based AntMiner algorithm, Weiss saw an improvement in the overall running time with the AntMinerGPU and ClusterFlockGPU had produced a slightly better clustering results when compared to the Data Swarm Clustering Algorithm. However, unlike the AntMinerGPU, the ClusterFlockGPU didn't show any improvements in running time. This suggests that not all Swarm Intelligence Algorithms would see a decrease in running time when implemented with a GPU-Based model but it may produce better results than a no GPU-Based model.

2.1.5 FASI

Li et al. (2018) had created a Framework for Accelerating Swarm Intelligence (FASI). FASI had reportedly achieved a better speedup on Field Programmable Gate Arrays (FPGA) when compared to other GPU-Based SIA algorithms. Li also reported that "FASI is up to 123 times and not less than 1.45 times faster in terms of optimization time on Xilinx Kintex Ultrascale xcku040 when compares to Intel Core i7-6700 CPU/ NVIDIA GTX 1080 GPU." Li also concludes that FASI reroutes the data flow of a SIA to prevent a bottleneck of parallelising SIAs "due to the heavy data transmission and the hierarchical memory architectures of the hardware platforms."

2.1.6 GPU-based swarm intelligence for Association Rule Mining in big databases

Djenouri et al. (2019) implemented a GPU-based model into the meta heuristic swarm intelligence, Bee Swarm Optimisation (BSO). This study was produced to reduce the time required for Association Rule Mining (ARM) task. ARM is a data mining task that is used to detect hidden patterns inside transaction databases. Results showed that the ALLGPU model had outperform the other 3 models used when comparing speed up.Djenouri et al. (2019) also showed that ALLGPU performed faster for low values with minimum support.

2.1.7 FastPSO: Towards Efficient Swarm Intelligence Algorithm on GPUs

Liu et al. (2021) had modified the PSO algorithm to test the performance against CPU-based PSO and GPU-based PSO. The results showed that their modified PSO, 'FASTPSO', had outperform both existing CPU and GPU model by 5 to 7x. With the modification, the FASTPSO also produced better optimisation results that the others

2.1.8 Dealing with Swarm Intelligence on GPUs

GPU-based MEtaheuristics Library (GMEL) is a library that utilises parallelism for swarm intelligence metaheuristics. It exploits parallelism of a GPU by using CUDA, which is a C/C++ programming language. They showed that the library applied to PSO, produced a satisfying speedups when running on simple fitness functions and the speed is better when applied to harder real-world problems.

2.1.9 A performance study of parallel programming via CPU and GPU on swarm intelligence based evolutionary algorithm

Lin & Phoa (2017) had analyzed the performance between CUDA and OPENMP with Swarm Intelligence Based Evolutionary (SIB-EA). CUDA was used for GPU programming and OPENMP was used to multi-process the SIB-EA. The results showed that OPENMP had an approximate speed up of 20x when compared to CUDA. CUDA had appeared to be inefficient as it had failed to reach a certain degree of parallelism. Lin & Phoa (2017) concluded with the promotion of OPENMP for parallel programming on evolutionary algorithms and states how CPU's shouldn't be left out on multi-processing.

3 Method

The goal of the is project is to modify a SI algorithm, permitting it to run on a GPU and to be multi-processed. The SI that will be chosen for this experiment will be Dispersive Fly Optimisation (DFO) as its new SI, which hasn't been modified to a GPU version or has gone through Multi-Processing. Making it a viable SI model to be used. From Tan & Ding (2015), there are 4 ways to go about modifying a Swarm Intelligence Algorithms (SIA) to a GPU-based version. The Naive Parallel Model, the Multiphase Parallel Model

3.1 Naive Parallel Model

Applying the Naive Parallel Model (NPM) to DFO shouldn't be difficult as only the fitness function would parallelised on the GPU, as shown in Figure 4. Tan & Ding (2015) stated that it could be applied to any SIA and speed up the execution time of the algorithm.So when applied to DFO, it would still be running through the normal serial manner. However, the fitness function would be parallelised on the GPU.

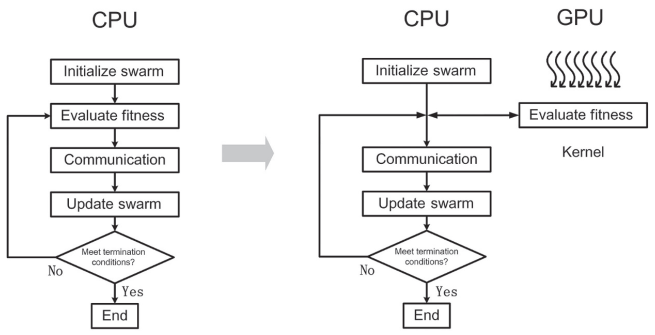

Figure 4: Naive Parallel Model Tan & Ding (2015)

3.2 Multiphase Parallel Model

Like the NPM, the Multiphase Parallel Model (MPM) also evaluates the fitness on the GPU. However, a feature in the SIA would also undergo parallelism(e.g. Figure 5). When applied to the DFO, updating the flies could be parallelised along with the fitness. However, communication between the host(CPU) and

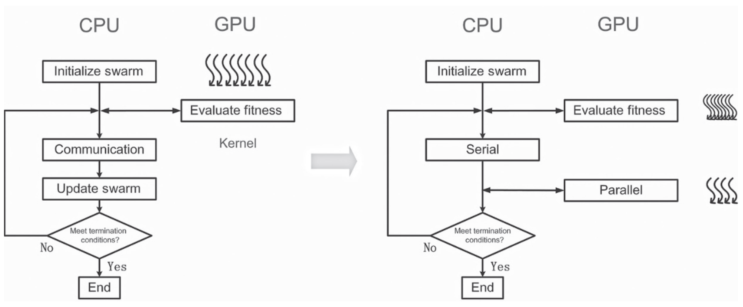

Figure 5: Multiphase Parallel Model Tan & Ding (2015)

device(GPU) when parallelising Fly Update would be slow. Knowledge on py-Cuda and C/C++ would be needed in order to make this modification possible.

3.3 All-GPU Parallel Model

The All-GPU Parallel Model (AGPM) is where the GPU is fully parallel and like the previous two, the swarm is initialized on the CPU. This makes up for the MPM, as their wouldn't need to be constant communication between the CPU and the GPU. This GPU would handle most of the computation and hopefully improve the speed of DFO execution

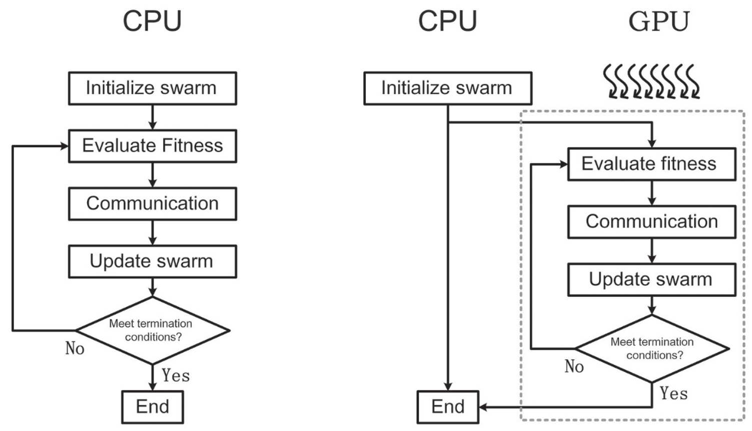

Figure 6: All-GPU Parallel Model Tan & Ding (2015)

3.4 Multiswarm Island Model

Finally, Multiswarm Island Model (MIM). The model also uses the GPU. However, unlike the previous models, the swarm is split up into 'islands' and sub swarms are formed. As presented in Figure 7. Each island is executed on a group of different threads in the GPU whilst retaining the ability of communicating with each other in an attempt of solving the same problem. Applying this model to the DFO would be would difficult to achieve and the communication area between flies would be heavily focused on.

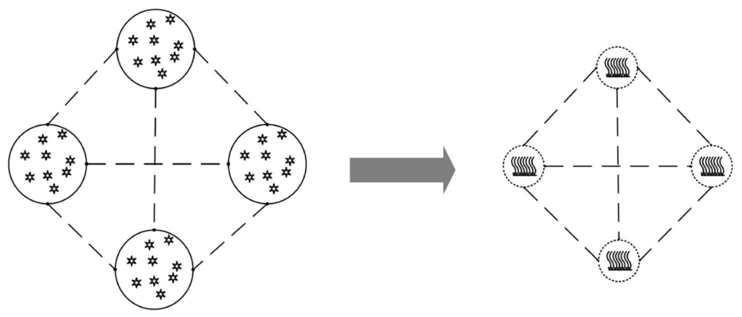

Figure 7: Multiswarm Parallel Model Tan & Ding (2015)

12 fitness function will be seached for and coded using the numpy library. These fitness functions would be a mixture of unimodals, low dimensionals and multimodals. An attempt to implement all of these models into DFO, will be difficult, but it would be attempted. Making a multi-process based DFO will also produced separately. The Multi-DFO (Multi-processed DFO) will be ran

on the CPU for the given amount of iterations. When Multi-DFO is executed, it will have to save its produced results to a text file. Upon completion, there would be an attempt multi-processing to a GPU-based model. This will confirm whether a combination of the two would have better performance. In both run time and fitness function results.

4 Experiment

To study the performance of between DFO, N-DFO and Multi-DFO, the equipment's used will be:

- OS - Windows 10 Home
- CPU - AMD Ryzen 5 2600X (3.60 GHz, 6 cores and 12 processors)
- RAM - 16GB
- GPU - NVIDIA GeForce RTX 3070 Ti (6144 CUDA cores)

To compare the performance between DFO and N-DFO, both will be ran 30 times against the benchmark functions below (Table 1).

Table 1: BENCHMARK FUNCTIONS

Fn	**Name**	**Class**	**Dimension**	**Feasible Bounds**
f_1	Sphere	Unimodal	30	$(-100, 100)^D$
f_2	Schwefel 1.2	Unimodal	30	$(-100, 100)^D$
f_3	Rastrigin	Multimodal	30	$(-5.12, 5.12)^D$
f_4	Rosenbrock	Multimodal	30	$(-2.048, 2.048)^D$
f_5	Ackley	Multimodal	30	$(-32.768, 32.768)^D$
f_6	Griewank	Multimodal	30	$(-600, 600)^D$
f_7	Lunacek's Bi-Rastrigin	Multimodal	30	$(-5.12, 5.12)^D$
f_8	Schaffer N0 6	Low Dimensional	2	$(-100, 100)^D$
f_9	Goldstein-Price	Low Dimensional	2	$(-2, 2)^D$
f_{10}	Six-Hump Camel	Low Dimensional	2	$(-5, 5)^D$
g_1	Shifted Rastrigin	Multimodal	30	$(-5, 5)^D$
g_2	Shifted Rosenbrock	Multimodal	30	$(-2, 2)^D$

Benchmark functions (also known as fitness functions) aid in determining the best fly in the swarm. As they would evaluate how close a fly would be towards the optimum solution. The experiment would also be using the libraries numpy, pyCuda, numba, Multi-Processing and pyTorch may be used. These libraries would aid in modifying DFO, to utilise Multi-Processing and make it GPU-based. CUDA run-time version is 11.6 but would be 10.2 if pyTorch is used.

The algorithm will be ran 30 times, with 3000 iterations being executed in each run. The parameters change depending on the fitness function being used, however, the population size N will be 100, the Δ (disturbance threshold) will be 0.001. To ensure that the experiments and results are fair, this will be applied to all modified (DFOs including the original)

5 Results

Firstly, DFO and the modified version are ran 30 times using the different benchmark functions from Table 1. This would shows whether the modified version would show better performance in both running time and results.

Fn	**DFO**		**N-DFO**		**Multi-DFO**		**Multi-N-DFO**	
	Mean	**StdDev**	**Mean**	**StdDev**	**Mean**	**StdDev**	**Mean**	**StdDev**
f_1	2.33e-29	6.95e-29	1.19e-29	3.34e-29	1.45e-29	2.34e-29	3.68e-30	4.83e-30
f_2	6.31e-25	1.53e-24	2.62e-25	4.99e-25	2.68e-25	5.16e-25	2.00e-24	9.96e-24
f_3	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00	0.00e+00
f_4	7.36e+00	5.18e+00	1.01e+01	3.82e+00	1.10e+01	1.13e+01	9.78e+00	4.13e+00
f_5	4.11e-14	9.02e-15	4.41e-14	9.89e-15	4.49e-14	1.28e-14	4.26e-14	7.03e-15
f_6	3.01e-02	3.21e-02	3.32e-02	3.39e-02	2.51e-02	2.40e-02	3.63e-02	3.10e-02
f_7	1.70e+01	1.49e+01	1.60e+01	1.50e+01	2.00e+01	1.41e+01	2.28e+01	1.28e+01
f_8	-9.36e-01	3.33e-16	-9.36e-01	3.33e-16	-9.36e-01	3.33e-16	-9.36e-01	3.33e-16
f_9	3.00e+00	1.39e-15	3.00e+00	1.24e-15	3.00e+00	1.42e-15	3.00e+00	1.19e-15
f_{10}	-4.00e+00	0.00e+00	-4.00e+00	0.00e+00	-4.00e+00	0.00e+00	-4.00e+00	0.00e+00
g_1	1.20e+02	4.80e+00	1.20e+02	6.48e+00	1.19e+02	6.60e+00	1.20e+02	5.52e+00
g_2	4.91e+00	4.09e+00	8.16e+00	3.02e+00	6.16e+00	3.85e+00	6.52e+00	3.79e+00

Table 2: After 30 Trials (to 3 significant figures)

From Table 2, it is shown that the results produced by the modified DFOs, do not vastly differ from the original DFO. Although the results did not greatly differ from each other, the running time of each function did.

Fn	DFO	N-DFO	Multi-DFO	Multi-N-DFO
f_1	1494	751.8	196.6	113.1
f_2	1049	772.0	132.0	112.7
f_3	1569	679.4	202.5	103.1
f_4	2882	719.4	365.1	102.6
f_5	1374	719.3	176.9	101.8
f_6	1685	720.6	223.1	113.7
f_7	1826	711.5	234.8	102.7
f_8	120.1	71.05	19.38	19.84
f_9	249.1	69.64	34.81	20.08
f_{10}	173.3	78.32	26.10	19.32
g_1	2275	912.3	306.5	112.6
g_2	3129	887.0	403.6	113.5

Table 3: Running Time (in seconds)

From 3, it is visible that Multi-N-DFO outperformed the other modifications and original by a vast amount. From Figure 8, N-DFO was 4x faster than the original and this was just achieved by evaluating the fitness on the GPU. When multi-processing is applied, with Multi-DFO, the algorithm runs at 7x the original. With the combination of the two, Multi-N-DFO beats the original by 28x the speed.

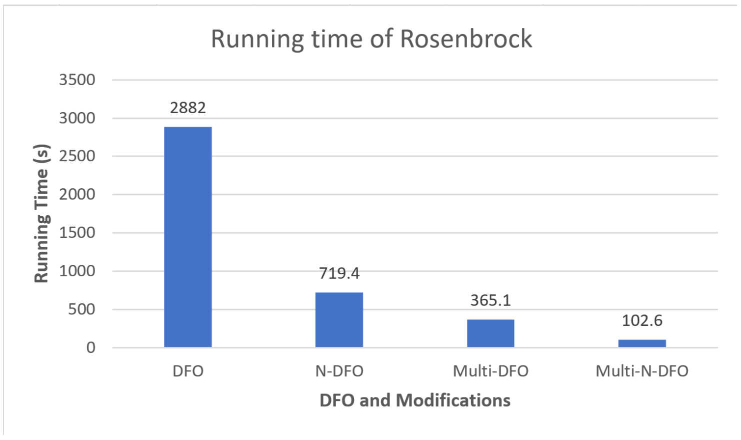

Figure 8: Rosenbrock Results

Although Multi-N-DFO beat the others in run time, it had failed in beating Multi-DFO. In Figure 9, Multi-N-DFO loses to Multi-DFO by 0.46 milliseconds. Although, this was the only case of loss, when closely looked at the rest of the Low Dimensional fitness functions. Multi-N-DF0 had struggled and barely beat

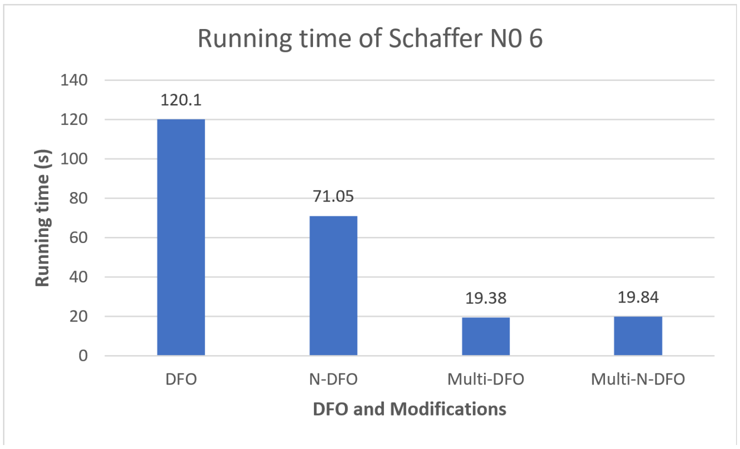

Figure 9: Schaffer N0 6 Results

Multi-DFO. Despite the struggle, all modified version still beat the original in run time.

6 Evaluation

From the result, GPU, Multi-Processing in swarm intelligence can be favourable due to its immense speed and results. Table 3 proves this, as it confers that a NPM-based DFO can shorten the run time of the algorithm by almost 4 times the original if not half. The multi-processed-based version had a tremendous run-time, even when compared to the GPU-Based version. Also, the multi-processed-based version had also produced better results that other versions in some cases. For example, f_6 function where, Multi-DFO had produced a mean of 2.51e-02 and standard deviation of 2.40e-02. It also performed better in f_4 with a mean and StdDev of 1.10e+01, 1.13e+01. Although, Multi-N-DFO had outperformed Multi-DFO in run time for both functions. Also, original DFO did get better results N-DFO. It was stated earlier that Multi-N-DFO had struggled to the Low dimensional function and even lost in one, it also struggled in unimodal f_2 function. It only beat the Multi-DFO model by 17.3 seconds. N-DFO also produced a better result than Multi-DFO in function f_2. Unfortunately, it was 5.8x slower than Multi-DFO. Based off this evaluation, it is safe to say that integrating multi-processing and GPU programming into SI has proven to be beneficial. It can both improve the performance in results and run time of the algorithm.

7 Conclusion

The aim of this project was to modify the SIA, DFO, into a GPU-Based Model and multi-process it. In this project it was done, however, only one GPU-based model was applied. An improvement that could be added in the future would be the implementation of all models of the GPU-based techniques and measure the results of both. Hopefully they follow the trend of Table 3 and it will show time ran in milliseconds. This supports Lin & Phoa (2017)'s take that CPU's shouldn't be left out in parallel programming as they can provide a great performance. It proved difficult to integrate such models as it would require a substantial understanding of C/C++ when GPU programming with CUDA, which is currently not possessed. An attempt was made on MPM in file MultiPhase.py. The attempt clearly failed and further research would need to be done. Another implementation that would be favoured, is the FASI model. If this could be applied to DFO, it will confirm whether the framework would beat GPU-based DFO.

References

Al-Rifaie, M. M. (2014), Dispersive flies optimisation, *in* '2014 Federated Conference on Computer Science and Information Systems', IEEE, pp. 529–538.

Beni, G. & Wang, J. (1993), Swarm intelligence in cellular robotic systems, *in* 'Robots and biological systems: towards a new bionics?', Springer, pp. 703–712.

Boiani, M. & Parpinelli, R. S. (2020), 'A gpu-based hybrid jde algorithm applied to the 3d-ab protein structure prediction', *Swarm and Evolutionary Computation* **58**, 100711.

Catala, A., Jaen, J. & Mocholi, J. A. (2007), Strategies for accelerating ant colony optimization algorithms on graphical processing units, *in* '2007 IEEE Congress on Evolutionary Computation', IEEE, pp. 492–500.

Darwin, C. (1859), *The origin of species by means of natural selection*, Pub One Info.

Ding, K., Zheng, S. & Tan, Y. (2013), A gpu-based parallel fireworks algorithm for optimization, *in* 'Proceedings of the 15th annual conference on Genetic and evolutionary computation', pp. 9–16.

Djenouri, Y., Fournier-Viger, P., Lin, J. C.-W., Djenouri, D. & Belhadi, A. (2019), 'Gpu-based swarm intelligence for association rule mining in big databases', *Intelligent Data Analysis* **23**(1), 57–76.

Dorigo, M. & Di Caro, G. (1999), Ant colony optimization: a new meta-heuristic, *in* 'Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)', Vol. 2, IEEE, pp. 1470–1477.

Fogel, D. B. (2006), *Evolutionary computation*, John Wiley Sons.

Holland, J. H. (1992), 'Genetic algorithms', *Scientific American* **267**(1), 66–73.
URL: *http://www.jstor.org/stable/24939139*

Kennedy, J. & Eberhart, R. (1995), Particle swarm optimization, *in* 'Proceedings of ICNN'95 - International Conference on Neural Networks', Vol. 4, pp. 1942–1948 vol.4.

Li, D., Huang, L., Wang, K., Pang, W., Zhou, Y. & Zhang, R. (2018), 'A general framework for accelerating swarm intelligence algorithms on fpgas, gpus and multi-core cpus', *IEEE Access* **6**, 72327–72344.

Li, J., Wan, D., Chi, Z. & Hu, X. (2006), 'A parallel particle swarm optimization algorithm based on fine-grained model with gpu-accelerating', *Journal of Harbin Institute of Technology* **38**(12), 2162–2166.

Lin, F. P.-C. & Phoa, F. K. H. (2017), A performance study of parallel programming via cpu and gpu on swarm intelligence based evolutionary algorithm, *in* 'Proceedings of the 2017 International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence', pp. 1–5.

Liu, H., Wen, Z. & Cai, W. (2021), Fastpso: Towards efficient swarm intelligence algorithm on gpus, *in* '50th International Conference on Parallel Processing', ICPP 2021, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3472456.3472474*

Moore, G. (2015), Moore's law, *in* 'GORDON MOORE: THE MAN WHOSE NAME MEANS PROGRESS', IEEE Spectrum.

NVIDIA (2015), 'Cuda c++ programming guide', *Guides* .
URL: *https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf*

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. & Phillips, J. C. (2008), 'Gpu computing', *Proceedings of the IEEE* **96**(5), 879–899.

Roser, M. & Ritchie, H. (2013), 'Technological change', *Our World in Data* . https://ourworldindata.org/technological-change.

Rotman, D. (2020), 'We're not prepared for the end of moore's law', *MIT Technology Review* .

Sawai, H. & Kizu, S. (1998), Parameter-free genetic algorithm inspired by "disparity theory of evolution", *in* A. E. Eiben, T. Bäck, M. Schoenauer & H.-P. Schwefel, eds, 'Parallel Problem Solving from Nature — PPSN V', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 702–711.

Tan, Y. & Ding, K. (2015), ‘A survey on gpu-based implementation of swarm intelligence algorithms’, *IEEE transactions on cybernetics* **46**(9), 2028–2041.

Weiss, R. M. (2010), GPU-accelerated data mining with swarm intelligence, PhD thesis, Honors Thesis, Department of Computer Science, Macalester College.

Yang, X.-S. & Deb, S. (2009), Cuckoo search via lévy flights, *in* ‘2009 World congress on nature & biologically inspired computing (NaBIC)’, Ieee, pp. 210–214.