A Performance Study of Parallel Programming via CPU and GPU on Swarm Intelligence Based Evolutionary Algorithm

Frank Po-Chen Lin
College of Electrical Engineering & Computer Science
National Taiwan University
EE Building No 2, 1 Roosevelt Road Section 4
Daan District, Taipei 106, Taiwan
(886) 2-3366-3700
r05942033@ntu.edu.tw

Frederick Kin Hing Phoa
Institute of Statistical Science
Academia Sinica
128 Academia Road Section 2,
Nangang District, Taipei 115, Taiwan
(886) 2-6614-5634
fredphoa@stat.sinica.edu.tw

ABSTRACT
Algorithm parallelization diversifies a complicated computing task into small parts, and thus it receives wide attention when it is implemented to evolutionary algorithms (EA). This works considers a recently developed EA called the Swarm Intelligence Based (SIB) method as a benchmark to compare the performance of two types of parallel computing approaches: a CPU-based approach via OpenMP and a GPU-based approach via CUDA. The experiments are conducted to solve an optimization problem in the search of supersaturated designs via the SIB method. Unlike conventional suggestions, we show that the CPU-based OpenMP outperforms CUDA at the execution time. At the end of this paper, we provide several potential problems in GPU parallel computing towards EA and suggest to use CPU-based OpenMP for parallel computing of EA.

CCS Concepts
• Computing methodologies → Optimization algorithms and Massive parallel algorithms • Software and its engineering → Distributed memory and parallel programming languages.

Keywords
Swarm Intelligence; Parallel Computing; OpenMP; CUDA.

1. INTRODUCTION
Evolutionary algorithm (EA) is typically a population based stochastic search technique and has been successfully used to solve hard optimization, search, and machine learning problems. It achieves a high level of problem solving efficacy in many engineering application areas, such as civil, mechanical, and industrial engineering, computer science, power systems, control, and signal processing in the engineering [1] or in the area of biomedicine/bioinformatics for cancer chemotherapy optimization, cancer chemotherapy drug scheduling model development and problem solving for protein folding [2]. As the amount of processing data have become nearly inconceivable, EAs have been applied to solve optimization problems with increasing difficulty and complexity [3]. The sequential programming no longer suffices for the needs of EAs. In order to improve the efficiency of EAs, parallel implementations have been used to significantly reinforce and speed up the search, allowing to achieve high quality results in reasonable execution times [4].

Parallel computing is a type of computation that simultaneously utilizes multiple computing resources (such as cores, computers) to solve a computational problem. A parallel programming is created for performing the normally sequential steps of a computer program simultaneously, using two or more processors. CPUs and GPUs have significantly different architectures that make them better suited to different tasks. Multitasking is heavily dependent on the type of application and since it could be sequential or parallel, CPU and GPU both are essential to perform better on such cases. Recently, much research focuses on expanding the usage scenarios for GPU since it works well to use large scale data decomposition and offers orders of magnitude speedups on those problems with highly parallel structure [5], [6]. However, individual processing units in a GPU cannot match a CPU for general purpose performance for that they are much simpler and do not have optimizations like long pipelines, out-of-order execution and instruction-level-parallelization. In addition, GPU computing also requires data transfer between CPU and GPU and cause data transmission overhead. [7] has also mentioned some of the problem in GPU programming.

This study explores the extent to which traditionally CPU domain problems can be mapped to GPU architectures using current parallel programming models and provide insights into why certain throughput computing kernels perform better on CPU and others work better on GPU. GPU performance is compared to both single-core and multi-core CPU performance. For GPU computing, [8] and [9] have found the overall performance of general purpose GPU much slower considering data transfer latency and lay stress on the necessity of including memory transfer overhead when reporting GPU performance. We present different parallel models, Open Multi-Processing (OpenMP) and Compute Unified Device Architecture (CUDA), on the swarm intelligence based-evolutionary algorithm (SIB-EA) [10] to analyze the performance of parallel implemented swarm intelligence based Supersaturated Design (SIBSSD) algorithm between CPU and GPU as our benchmark of EAs. Finally, we evaluate their efficiencies under different implementations via their simulation results.

This paper is organized as follows. Section 2 presents the two main computing procedures on SIB-EAs – CPUs and GPUs with OpenMP and CUDA respectively. Section 3 demonstrates the simulation results with SIBSSD. Finally, concluding remarks are presented in Section 4.

ISMSI '17, March 25–27, 2017, Hong Kong, Hong Kong.

DOI: http://dx.doi.org/10.1145/3059336.3059339

2. Parallel Computing: CPU versus GPU

The aim of this study is to compare the efficiency of parallel programming between OpenMP, CUDA and a single-core CPU to explore the efficiency to implement SIB-EAs with parallel programming. In the next section, we run SIBSSD on different platforms using OpenMP, CUDA and single-core CPU respectively.

2.1 Swarm Intelligence Based Evolutionary Algorithm (SIB-EA)

Many researches are interested in the analysis of big data and emphasize the importance of computational speed when dealing with the data calculation. In this paper, instead of only focusing on the speed of parallel computing, we are also interested in the availability of parallel programming when dealing with EAs since a large number of their functionalities are distinct and self-contained. We analyzed the core computation and coding characteristics of SIB- EA kernels using both OpenMP and CUDA.

The common idea in EAs behind all the techniques are the same: given a group of randomized particles as a set of candidate solutions and apply an objective function as an abstract fitness measurement. Based on this fitness, the suitable candidates are chosen to seed the generation by applying mutation (MIX) and recombination (MOVE) to them. After executing MIX and MOVE procedure on the candidates, the candidates (the elements inside each particle) are evaluated by the objective function, the best-fitted candidates will be set as the updated particles and being selected as new candidates for the next generation in the next iteration. The process continues until either a candidate with sufficient quality is found or the computational limit is reached. Details are omitted but interested readers may refer to [11]. In this research, SIBSSD is executed in a parallel fashion as a benchmark of parallel implemented SIB-EAs. E(s^2) is used as the quality function, further detail for E(s^2) may refer to [12].

2.2 Computing with CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a NVIDIA GPU for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements and is designed to work with programming languages such as C and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources. To compute with CUDA, it is required to communicate between the HOST (CPU) and DEVICE (GPU). To start parallel programing in GPU, we have to first transfer the data from the HOST to the DEVICE, then we can start computing in the GPU with multiple threads. Finally, the data is transferred back to the HOST after the computations end. Our experiment runs on the GPU platforms, NVIDIA GeForce GT 740M with the process mainly executed in the kernel function of CUDA based on C.

The main idea of our computing kernel for an evolutionary algorithm like SIBSSD is discussed and shown in Figure 1. There are two main parts in the computing procedure, initialization and main operation. In the first part, the code is written in original C language in CPU end and that GPU has not yet been used. The procedure randomly generates a set of balanced $N \times m$ matrix as initial particles where each column in the matrix represents a single particle, evaluates the objective function (E(s^2)) value of each particle, initializes Local Best (LB) for each particle and Global Best (GB) for all the particles and sets the initialized particles as the candidate solutions.

During the iteration, MIX and MOVE operations are carried out. The MIX operation is a column exchange procedure, consisting of

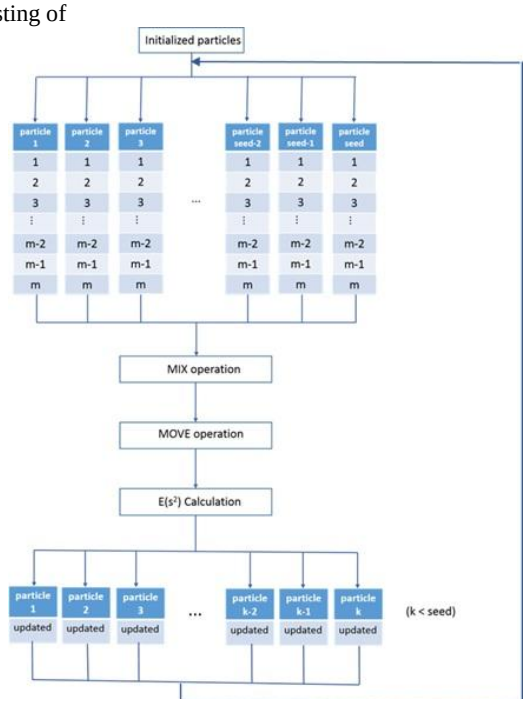

Figure 1. The process of SIBSSD with CUDA.

column deletion and addition. For each generated candidate i.e. $X = (x_1, \dots, x_m)$, q columns are replaced in X by q columns from another candidate $Y = (y_1, \dots, y_m)$, so that the replaced design has a smaller criterion value. In the MOVE operation, the movement of a particle is completed by replacing each current particle with possibly another particle to reach a smaller value of E(s^2).

Before starting the calculation in GPU, we use `cudaMalloc` and `cudaMemcpy` in the code for allocating the size of the memory and for transferring the data required for computation in the GPU. The particles are grouped together into blocks in CUDA, and blocks are processed in parallel. Then, the MIX operation mainly deals with deleting and adding columns, the particles are first transformed into matrices containing $\{-1,1\}$and operate the column deletion simultaneously. In the column deletion, the correlations between column pairs are computed and saved as correlation matrices, then we search and delete the index of columns with the largest correlation in the matrices. After the column deletion, the column addition follows, which does almost the same as deleting columns except that the particle transformation matrices are added with the columns on the index that has the minimum value of correlation.

In the MOVE operation, particles replace each current particle with another particle to reach a smaller value in the quality function. With the evaluated E(s^2) value of each particles in the MIX procedure, we compare the E(s^2) values among three options and decide which one should be chosen as the new candidate in the next iteration. In this part, if/else statements and index-searching are operated to compare the values of the particles simultaneously in each blocks.

In CUDA, memory allocation is also an important issue of speeding up the procedure in an efficient way. The memory

architecture of the NVIDIA GPU is depicted as Figure 2. In the HOST end, the study sets the initialized particles in the global memory instead of the constant memory for that the constant memory has a size limit of 64KB that is not enough to save the entire data when we have particles samples larger than 64KB, which is often the situation during big data analysis even though the constant memory has a cache with 8KB but the global memory does not. On the other hand, as a profitable way of performing computation on the device, we partition the particles as a data subset and load it into the declared shared memory to take advantage of its fast accessing speed. Share memory is a low-latency, high-bandwidth, indexable memory that runs close to register speeds. Therefore, when we compute on the subset from shared memory, each thread can access the data efficiently with the highly decreased latency overhead with every threads in the same block sharing the same memory.

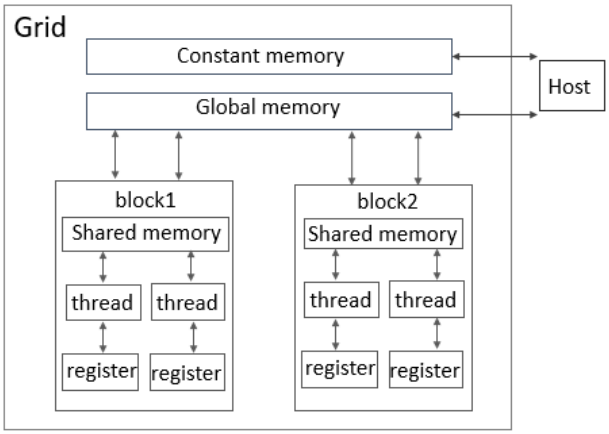

Figure 2. Memory architecture of GPU with CUDA.

2.3 Computing with OPENMP

OpenMP is an implementation of multithreading, a method of parallelizing a master thread into a specified number of slave-threads to divide the task among them, depicted in Figure 3. The threads then run concurrently, with the runtime environment allocating threads to different processors. This study uses Intel(R) Core(TM) i5-3337U CPU (Dual Core) with 1.8GHz clock speed as our platform for OpenMP to execute the SIB-EA, choosing the same section to parallelize as we did in CUDA. By using the APIs provided by OpenMP, this study parallelizes the for-loops inside the kernel by adding *#pragma omp parallel {#pragma omp for schedule(static), private(j)*, which is indeed quite simple when compared to CUDA. To ensure all threads complete their tasks before moving to the next iteration, we add *#pragma omp barrier}* at the end of the parallel section to pause all threads until all threads execute the barrier. Instead of parallelizing the particles, OpenMP parallelizes the for-loop inside the kernel shown in Figure 4. The main for-loop is executed simultaneously with the cores in the CPU.

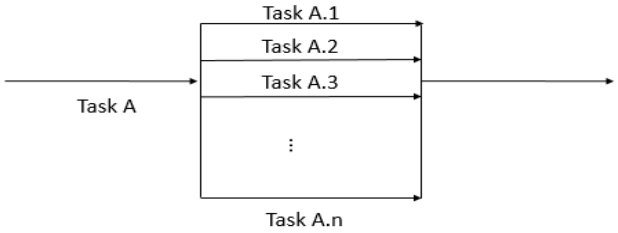

Figure 3. The process of SIBSSD with OpenMP.

3. Demonstration and Discussion : SIBSSD Example

In this section, the study evaluates the performance difference between CUDA and OpenMP by testing the execution time of SIBSSD on each platform, where SIBSSD is set as the benchmark for SIB-EAs. This paper takes the data transfer time into account for fair comparison of real speedup provided by a device.

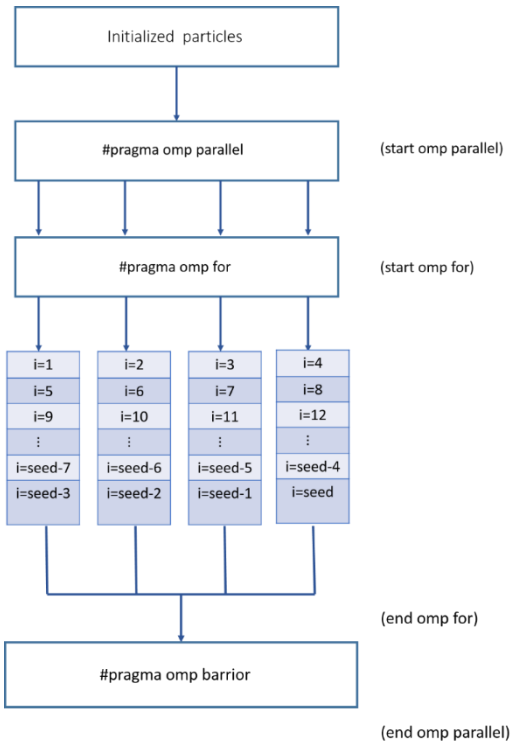

Figure 4. Loop inside the kernel of OpenMP.

The total execution time is analyzed and it is set as an indicator to observe the computing performance. This paper compares the difference of the execution time with different parameter values in SIBSSD with $seed \times m$ SSDs, where m is the size of the particle and $seed$ is the total number of SSDs. The size of the input data has a major influence on the execution time of the programs. The difference between the time span of the execution is discussed by setting the parameters values as the control variable and we focus only on the performance between CPU and GPU.

Visual Studio 2013 is used to conduct the comparison between CUDA and OpenMP under several premise. The computation limit of $E(s^2)$ value and the total iterations are set to be 2.5 and

100 respectively. This study adopts NVIDA CUDA toolkit 7.5 for operating CUDA in Visual Studio 2013.

3.1 Performance Comparison of SIBSSD

We execute SIBSSD in two scenarios: CUDA and OpenMP (dual core). The execution results under different values of m and *seed* are demonstrated in Figures 5 and 6. The values on each bar of these figures are in the unit of seconds per iteration (s/iteration). An interesting observation is found in the figure: Computing in a single-core CPU is even more efficient than CUDA. As the dimensions grow, the exceeding performance of single-core CPU to CUDA becomes apparent. Thus, SIB-EAs are apparently not suited to execute in CUDA. CUDA has a less efficient performance compare to a single-core CPU. Second, the efficiency of OpenMP outperforms that of CUDA. In almost all cases, the operation time of OpenMP is about 20 times faster than that of CUDA. Apparently, SIB-EAs and many evolutionary algorithms are not suited to execute in CUDA. It is totally opposite to conventional suggestion about the promotion on the use of GPU parallel computing.

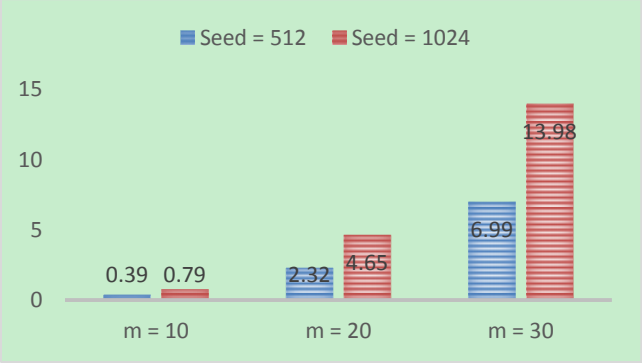

Figure 5. Runtime of SIBSSD with CUDA.

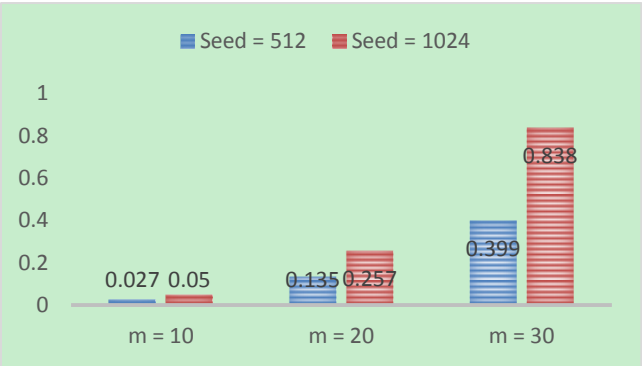

Figure 6. Runtime of SIBSSD with OpenMP.

3.2 Discussion

Figure. 7 shows the computing results that OpenMP has approximate a 20 speedup rate when compared to CUDA. In this discussion, we attempt to explain the underlying reasons why such unexpected phenomenon may result.

In SIBSSD and many evolutionary algorithms, there are some disadvantages of using CUDA as a computing platform. During the entire process in the kernel function, particles were parallelized into different share memory while each share memory consists of a particle with size of m and each block contains one share memory.

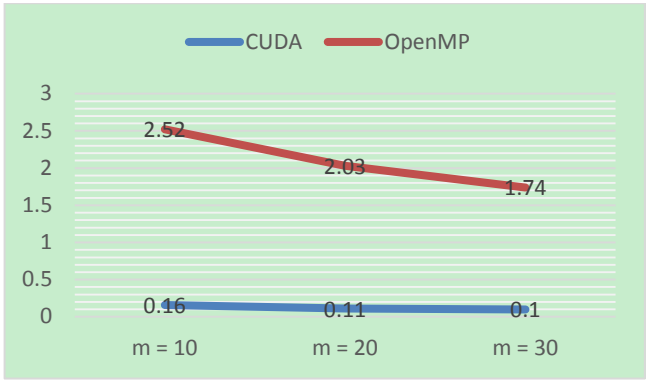

Figure 7. Comparison with CUDA and OpenMP.

It allows all the particles to operate the MIX and MOVE operations simultaneously with the particles parallelized in a block-dimension. However, in SIB-EAs, although the particles operate in the same way, each elements does not perform the same calculations. To follow the SIMD structure, all the calculations of each particle can only be carried out by a single thread in the GPU and makes it unable for the operation to achieve highly data-parallelism. In other words, the structure in SIBSSD does not allow us to make full use of the parallelization structure in GPU. Instead of parallelizing the elements of each particle into threads, the EAs only allows us to parallelize the particles into blocks. Consequently, GPU randomly chooses one thread to finish the entire operation for every cells in each particle, all the operations for the elements in the particles are operated with one GPU thread only and it causes an apparent increase in the operation time compare to CPU. Since the elements in each particles do a lot of linear algebra calculations with only one of the weak cores in GPU, the operation would be eventually more time-consuming.

On the other hand, for the MOVE operation, since the decision making step consists of a bunch of if/else branching, GPU computes both directions of the branch for all warps in each block and results in overhead and efficiency reduction.

In addition, most GPU are designed for stream or throughput computing, where cache memories are ineffective. To tolerate memory latency, it requires a high degree of multithreading. Nevertheless, the structure of SIBSSD is not able to be highly parallelized and it leads to a great reduction in the computing performance. Besides, we also take the time of data transfer into account for that we have to transmit the huge amount of data from CPU to GPU and then transfer the data back after the calculations in the kernel function. In brief, the EAs does not provide a highly parallel environment for all the elements in the particles to follow a SIMD model structure and would largely decrease the speed of calculation.

For OpenMP, both task parallelism and data parallelism can be achieved. It is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads and the threads can cache their data and are not required to maintain exact consistency with real memory all of the time, which can reduce the data accessing latency automatically without giving any other instructions to the computer. The caches provided by the CPU makes it easy for OpenMP to achieve task parallelism without latency overhead which is not the case in GPU due to its architectural implementation.

In SIBSSD, each thread in CPU parallelizes the candidate-updating iteration. Since the main iteration is executed

simultaneously and the operations of each particle are carried out by a much more powerful CPU core, OpenMP stands out in the performance. It is also interesting to know that OpenMP provides a user-friendly interface and would be easier for programmers to implement than CUDA.

Similar phenomenon is expected to happen in other EAs. These "nature-inspired" EAs feature inter-unit "communications", which usually lead to many if/else branching for unit comparison. In addition, their convergence towards optima require an attractive force from their overall best unit, which require excessive data transfer and increase calculation time. There are some EAs that do not need such attractive force, like genetic algorithm, but their convergence towards optima is usually slow.

4. CONCLUSIONS

As shown in the simulation, OpenMP (dual Core) has a better performance than CUDA (a 20-times speed increase on SIB-EAs).

For SIB-EAs, there are some disadvantages of using CUDA as a computing platform. For GPU to operate efficiently and faster than CPU, algorithm must take advantage of the large parallelism inherent in a GPU's design, particularly for floating point. In other words, the architecture feature of the algorithm is a key to accelerate the entire throughput computing workload. However, due to the structure of SIB-EAs, it fails to make full use of the parallelization structure, because the particles are only able to be parallelized into blocks consisting of only one thread. Since every thread is carried out by a weaker core than CPU, the computational workload increases. Besides, memory latency also reduces the computing efficiency in the benchmark of this study. Unlike OpenMP, the cache memory in most GPU are ineffective and mostly result in memory latency overhead if the operations do not reach a certain amount of parallel degree. In addition, the code for SIB-EAs consists of many if/else branching, so a performance reduction is resulted as the hardware computes both directions of the branch for all elements in a warp. Finally, there is data transfer time between CPU and GPU in CUDA. Similar phenomenon is expected to happen in many other EAs.

This paper concludes that the performance on the implementation of SIB-EAs using OpenMP highly stands out compared with CUDA and would be easier to implement. As a result, we promote OpenMP for parallel programming on evolutionary algorithms and emphasize the significance irreplaceable nature of CPU on parallel programming.

5. ACKNOWLEDGEMENT

The authors would thank Mr. Shyh-Kae Chou for his help in CUDA and OpenMP parallel computing. This work was supported by Career Development Award of Academia Sinica (Taiwan) grant number 103-CDA-M04, and Ministry of Science and Technology (Taiwan) grant numbers 104-2118-M-001-016-MY2 and 105-2118-M-001-007-MY2.

6. REFERENCES

[1] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of evolutionary computation*. Oxford University Press, 1997.

[2] C. C. Kannas, C. A. Nicolaou, and C. S. Pattichis, "A Parallel Implementation of a Multi-objective Evolutionary Algorithm," in *Proc. 9th International Conf. on ITAB*, pp. 1–6, Nov. 2009.

[3] S. Iturriaga, and S.Nesmachnow, "Solving very large optimization problems (up to one billion variables) with a parallel evolutionary algorithm in CPU and GPU," in *Proc. Seventh International Conf. on 3PGCIC*, pp. 267–272, Nov. 2012.

[4] E. Alba, *Parallel Metaheuristics*: A New Class of Algorithms. Wiley-Interscience, 2005.

[5] P. Emeliyanenko, "Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra," J. *Parallel and Distributed Computing*, vol. 73, no. 11, pp. 1494–1505, Nov. 2013.

[6] G.Makey and M. S. El-Dasher "Modification of common Fourier computer generated hologram's representation methods from sequential to parallel computing," *Optik - International J. Light and Electron Optics*, vol. 126, no. 11–12, pp. 1067–1071, June 2015.

[7] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J.C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[8] C. Gregg, K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in Proc. *IEEE International Symposium on ISPASS*, pp. 134–144, Apr. 2011.

[9] M. H. Rahmad, S. M. Meng, E. K. Karuppiah, and O. Hong, "Comparison of CPU and GPU implementation of computing absolute difference," *IEEE International Conf. on ICCSCE*, pp. 132–137, Nov. 2011.

[10] F. K. H. Phoa, R. B. Chen, W. Wang, and W. K. Wong, "Optimizing Two-level Supersaturated Designs using Swarm Intelligence Techniques," *Technometrics*, Oct. 2014.

[11] V. Roberge, M. Tarbouchi, and F. Okou, "Strategies to Accelerate Harmonic Minimization in Multilevel Inverters Using a Parallel Genetic Algorithm on Graphical Processing Unit," *IEEE Trans. on Power Electronics*, vol. 29, no. 10, pp. 5087–5090, Oct. 2014.

[12] D. Zubanovic, A. Hidic, A. Hajdarevic, N. Nosovic, and S. Konjicija, "Performance Analysis of Parallel Master-Slave Evolutionary Strategies (μ,λ) Model Python Implementation for CPU and GPGPU," in *Proc. 37th International Convention on MIPRO*, pp. 1609–1613, May 2014.

[13] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing. 2nd ed. *Springer* , pp. 15–35, 2015.

[14] Booth, K.H.V., and Cox, D.R, Some systematic supersaturated designs. *Technometrics*, vol. 4, pp. 489–495, 1962.