Paper accepted by Int. J. Process Systems Engineering on 10 Feb 2015,
Submitted for final publication by Inderscience.

gpuMF: A Framework for Parallel Hybrid Metaheuristics on GPU with application to the Minimization of Harmonics in Multilevel Inverters

Vincent Roberge, Mohammed Tarbouchi and Francis Okou

Department of Electrical and Computer Engineering
Royal Military College of Canada
PO Box 17000, Station Forces
Kingston, ON K7K 7B4, CANADA
E-mail: vincent.roberge@rmc.ca
E-mail: tarbouchi-m@rmc.ca
E-mail: okou.aime@rmc.ca

Abstract: Metaheuristics are non-deterministic optimization algorithms used to solve complex problems for which classic approaches are unsuitable or unable to generate satisfying solutions in a reasonable time. Despite their effectiveness, metaheuristics require considerable computational power. Multiple efforts have been made on the development of parallel metaheuristics on graphics processing units (GPUs). Based on a massively parallel architecture, the GPU offers remarkable computing power and can provide significant speedup. However, there currently exists no software project that unites these research initiatives into a comprehensive and reusable tool. To address this shortcoming, we developed *gpuMF*, a framework for parallel hybrid metaheuristics on GPUs. *gpuMF* (or GPU Metaheuristic Framework) exploits the intrinsic parallelism found in metaheuristics and fully utilizes the massively parallel architecture of GPUs. To demonstrate the effectiveness of our framework, we use *gpuMF* to minimize the harmonics of multilevel inverters while providing a speedup of 276x.

Keywords: software framework, metaheuristic, parallel computing, GPU, harmonic minimization, multilevel inverters.

Published version available online at Inderscience: Roberge, V., Tarbouchi, M. and Okou, F. (2015) 'gpuMF: a framework for parallel hybrid metaheuristics on GPU with application to the minimisation of harmonics in multilevel inverters', Int. J. Process Systems Engineering, Vol. 3, Nos. 1/2/3, pp.20–41.

Biographical notes: Vincent Roberge received his B.Eng. and M.A.Sc. from the Royal Military College of Canada (RMCC), Kingston, ON, Canada, in 2005 and 2010. In 2011, he joined the Department of Electrical and Computer Engineering at RMCC where he is currently Lecturer and PhD candidate. His current research interests include parallel computing, GPUs, optimization and smart-grids.

Dr. Mohammed Tarbouchi received his M.Sc. and Ph.D. from Laval University, Quebec, Canada in 1993 and 1997, respectively. In September 1997 he joined the Department of Electrical and Computer Engineering at the Royal Military College of Canada where he is currently Associate Professor. His current research interests include analysis and control of electrical machines, variable speed drives and real-time applications in smart-grids.

Dr. Okou received the Diplôme d'Ingénieur Degree in electrical engineering from École Supérieure Interafricaine de l'Électricité, Côte d'Ivoire, in 1993, the M.Ing. degree and the Ph.D. in electrical engineering from École de Technologie Supérieure (ÉTS), Montreal, QC, Canada, in 1996 and 2001, respectively. In 2005, he joined The Royal Military College of Canada. He is currently an associate Professor in the Electrical and Computer Engineering Department. His research interests include the application of advanced control techniques to power systems and electric drives.

1 Introduction

Metaheuristics have proved effective to optimize a multitude of complex problems for which there is no known classical method or that simply cannot be solved in a reasonable time by conventional methods. Metaheuristic algorithms are robust, efficient and simple to implement. However, despite their ability to optimize complex problems, they can require significant computing power and their execution time may remain too long for time critical applications. In fact, metaheuristics do not solve the problem directly, but use an iterative process to improve candidate solutions before reaching a quasi-optimal one. Fortunately, metaheuristics are inherently parallel and can be accelerated on a parallel system architecture. To deal with their high computational requirement, several parallel implementations have been proposed on multicore shared memory systems (Ragnarsson et al. 2011), computer clusters (Sena et al. 2001), supercomputers (Melab et al. 2006) and more recently on graphical processing units (GPUs) (Weyland et al. 2013). Affordable and installed in most desktop computers, GPUs are a very promising platform for parallel implementation of metaheuristics and can provide a significant speedup due to their hundreds or even thousands of cores. Many papers have been published in recent years on different parallelization strategies for metaheuristics on GPU, but there currently exist not software framework that gather those advancements into a comprehensive, reusable and extensible software tool. A programmers wanting to use a parallel metaheuristic on GPU for a specific application is left to re-implementing or coding a previously published solution from scratch.

In this paper, to address this shortcoming, we present *gpuMF*, a framework for parallel hybrid metaheuristics on GPUs. *gpuMF* (or GPU Metaheuristic Framework) is programmed in NVIDIA® CUDA™ C++ and exploits the intrinsic parallelism found in metaheuristics and fully utilizes the massively parallel architecture of GPUs. It offers a clear separation between the optimization tool and the problem considered allowing *gpuMF* to be reused for a wide range of optimization problems. Using an XML file, the user can easily select, configure and adapt the metaheuristics for a specific application. Moreover, to provide an optimization tool that is more robust against a wider range of problems, *gpuMF* supports hybrid metaheuristics based on the island model (Izzo et al. 2012) where the candidate solutions are divided into islands and each metaheuristic independently works on its respective island. Collaboration between the different metaheuristics is implemented by a migration process where selected solutions migrate from an island to another following different migration strategies offering a good balance between the exploration of the search space and convergence towards the final and quasi-optimal solution. Finally, to demonstrate the efficiency of metaheuristics and the speed improvement of our parallel implementation, we use *gpuMF* for the minimization of harmonics in multilevel inverters. Multilevel inverters form a popular class of high power

inverters due to their high voltage operation, high efficiency, low switching losses and low electromagnetic interference. Our proposed framework can compute the optimal switching angles for multilevel inverters with up to 100 dc sources while minimizing the first 100 harmonics. Compared to a sequential implementation, our parallel approach on GPU offers a speedup of 276x, allowing for more responsive control of the inverter.

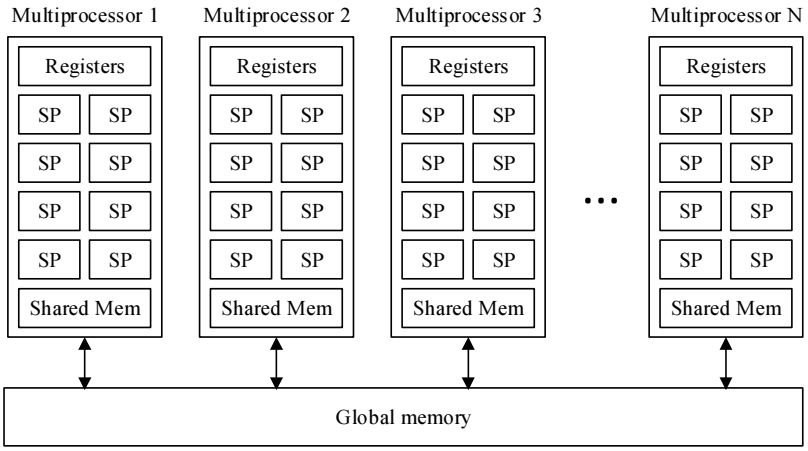
The reminder of this paper is organized as follows. In section 2, we describe the parallel architecture of GPUs and discuss some programming considerations. In section 3, we introduce the particle swarm optimizer (PSO). Although *gpuMF* uses more than one metaheuristic, discussing the PSO will allow the reader to better understand functioning of metaheuristics in general. In section 4, we review key contributions in the field of metaheuristics framework and identify the difficulty of porting current implementations to the GPU. In section 5, we present the design of our proposed framework *gpuMF*. Finally, to illustrate the speed advantage of the parallel framework on GPU, we show, in section 6, how *gpuMF* can be used to find the optimal switching angles to minimize the harmonics in a multilevel inverter.

2 Architecture of Graphics Processing Units

In recent years, graphics processors have seen their architecture change. The non-programmable parallel pipelines have been replaced by a large number of RISC processors supporting floating point operations. This change has allowed greater flexibility for graphics applications, but also the use of GPUs for scientific computing. Equipped with several hundred or even a few thousand of cores, GPUs provide computing power comparable to that of a computer network, but embarked on a single chip and present in most desktop computers. GPUs can be programmed for general computing using NVIDIA® CUDA™ since 2007 or OpenCL™ since 2008. Both are based on the C language, use reserved keywords for parallel sections and offer a set of API functions. CUDA is proprietary and only for NVIDIA GPUs while OpenCL is an open standard supported by many vendors including NVIDIA. In this researched, we used an NVIDIA GPU and therefore the CUDA language in order to achieved the best performance. Exploiting the high processing power of GPUs can be very beneficial for large problems in time critical applications, but requires important development effort. In order to understand the technical challenges in designing parallel software for GPUs, it is important to have a basic understanding of its parallel architecture. In this section, we describe the parallel architecture of GPUs and discuss some programming considerations.

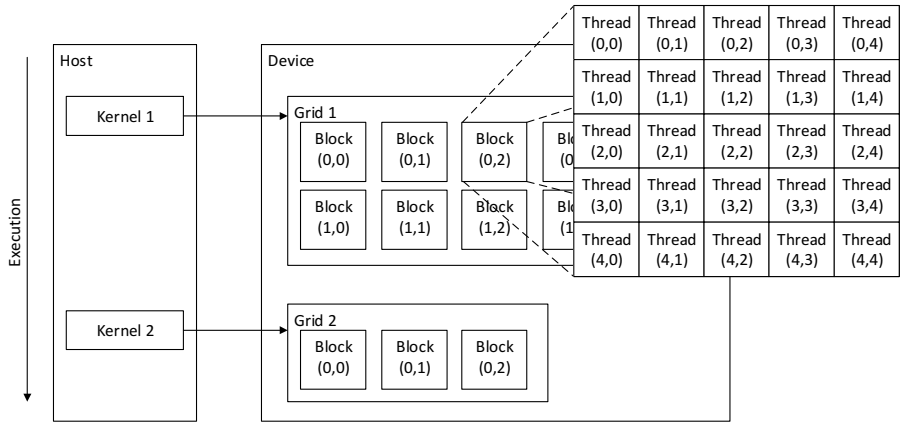
The typical architecture of NVIDIA graphics processors is show in Figure 1. These processors are composed of a large number of cores called *streaming processors* (SP) grouped into *multiprocessor* (MP) units. Each SP includes an arithmetic logical unit supporting integer and floating point operations. Although there can be 32 to 192 SPs per MP, the SPs share only a few instruction buffers. MP has a bank of registers and a low latency and high throughput shared memory. The GPU has a large off-chip random access memory shared by all MPs. The GPU interfaces with the CPU using the PCI Express (PCIe) bus. In the experimental tests discussed in section 6, we use the NVIDIA GTX 750 Ti SC card. This card has a GM107 chip, 640 SPs, 5 MPs, 64 K 32-bit registers per MP, 64 KB of shared memory per MP and 2 GB of DDR5 random access memory. The SPs operates at 1255 MHz and the global memory, at an effective frequency of 5400 MHz delivering a bandwidth of 86.4 GB/s.

Figure 1. Typical architecture of NVIDIA graphics processors



To program the GPU for general computation, one must write parallel functions called *kernels*. The execution model of a CUDA program is shown in Figure 2. The program always starts on the CPU (or host) and offload heavy calculations to the GPU (or device) by calling the *kernels*. A *kernel* launches a *grid* of threads organized into *thread blocks*. The *thread blocks* are mapped to the different MPs. Registers are private to a thread and cannot be shared. However, threads within the same block can cooperate by sharing data through the shared memory. CUDA provides synchronization operations within *thread blocks*, but not between *thread blocks*. The latter can be achieved by dividing parallel functions into multiple *kernels* since CUDA ensures that all *thread blocks* terminate before the next *kernel* is launched.

Figure 2. Execution model of a CUDA program



Developing parallel programs in CUDA can be challenging. To achieve the best performance, one must adapt the design to the architecture of the GPU by considering the following guidelines:

1) **Maximize data parallelism.** To benefit from the high computing power of the GPU, it is necessary that the parallel application uses a very large number of threads to maximize SPs utilization.
2) **Minimize thread divergence.** Although MPs are composed of multiple SPs, they have very few instructions buffers. The multiprocessor can therefore execute multiple threads simultaneously provided they all perform the same instruction. It is therefore best to avoid conditional statements that would result in divergent path as only one path can be executed at a time.
3) **Minimize data transfers on PCIe bus.** Since the GPU and the CPU have different memory spaces, the data must be copied over the PCIe bus. To avoid delays, data copies between the CPU and the GPU should be limited.
4) **Maximize coalescing access to the global memory.** The cache line and data bus of the GPU global memory are very large and allow memory access of multiple threads to be grouped into a single operation to deliver an effective bandwidth above 300 GB/s (much higher than that of a CPU). However, to achieve this performance, consecutive threads must access memory locations that are collocated (or coalesced).
5) **Maximize use of shared memory.** The shared memory is limited to 64 KB per MP, but has a latency 20 times smaller than the global memory. When accessing the same data multiple time inside a *kernel*, it is advantageous to use the shared memory as a user control cache or scratch pad memory.
6) **Maximize *multiprocessor* occupancy.** Each MP has 32 to 192 SPs, but enough registers to hold 2,048 resident threads. Maximizing the number of resident threads gives the MP room to reschedule threads and hide access latency to global memory. Unlike for typical CPU, CUDA thread scheduling is done in hardware and consumes no clock cycle.

Finally, there is also a final consideration to follow that is not necessarily specific to the GPU, but all parallel applications. Indeed, Amdahl's Law (Rauber & Rünger 2010) points out that the maximum speedup of a program is limited by its sequential part. As an example, let's take a sequential program that takes 1 minute to execute and where 20% of its execution time is intrinsically sequential and cannot be parallelized. Even if the other 80% was infinitely reduced by a parallelization, this program would still take 12s (due to the sequential part) and the speedup would be limited to 5x. During the development of parallel algorithms, it is therefore crucial to limit the sequential part and parallelize all stages of the algorithm. In this section, we have presented the architecture of GPUs and discussed key programming considerations. This will be useful in sections 4 and 5 when we review recent works and present *gpuMF*, our proposed framework for parallel metaheuristics on GPU.

3 Metaheuristics

In this section, we introduce the particle swarm optimizer (PSO). Although the proposed framework *gpuMF* implements more than one metaheuristic, discussing the PSO will allow the reader to better understand the basic functioning of metaheuristics in general. The PSO is a population based non-deterministic optimization method that was proposed by Kennedy and Eberhart in 1995 (Kennedy & Eberhart 1995). The algorithm simulates the movement of a swarm of particles in a multidimensional search space progressing towards an optimal solution. The position of each particle represents a candidate solution and can

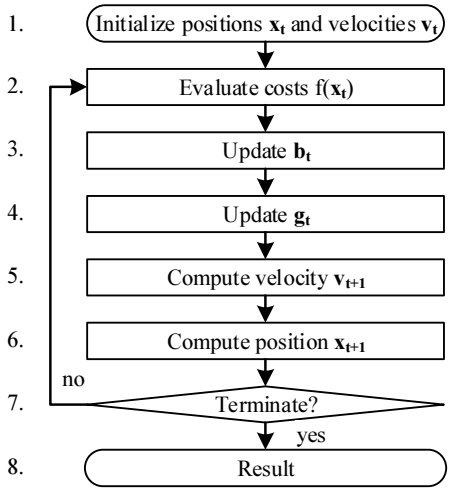
be randomly initiated. At every step of the iterative process, the velocity of each particle is individually updated based on the previous velocity of the particle, the best position ever occupied by the particle (personal influence) and the best position ever occupied by any particle of the swarm (social influence). As outlined in (Talbi 2009), the equations used to compute the velocity and position of a single particle at iteration t are as follows:

$$\boldsymbol{v}_{t+1} = \omega \boldsymbol{v}_t + c_1 \boldsymbol{r}_1 .* (\boldsymbol{b}_t - \boldsymbol{x}_t) + c_2 \boldsymbol{r}_2 .* (\boldsymbol{g}_t - \boldsymbol{x}_t) \quad (1)$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{v}_{t+1} \quad (2)$$

where variables in bold are vectors; $\boldsymbol{v}$ is the velocity of the particle; $\boldsymbol{x}$ is its position; $\boldsymbol{b}$ is the best position previously occupied by the particle; $\boldsymbol{g}$ is the best position previously occupied by any particle of the swarm; $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ are vectors of random values between 0 and 1; and ω, c_1 and c_2 are respectively the inertia, the personal influence and the social influence parameters. The flow diagram of the PSO is displayed in Figure 3.

Figure 3. Flow chart of the Particle Swarm Optimizer



Metaheuristics are general purpose optimization algorithms and are not equally effective against all optimization problems. Often referred to as the "No Free Lunch" theorem (Wolpert & Macready 1997), each metaheuristic exhibits specific strengths and must be carefully chosen and adapted to a given problem. In order to improve the robustness of the optimization algorithm, researchers have developed hybrid methods by mixing two or more metaheuristics. One approach to developing hybrid metaheuristic consists of mixing parts of two established algorithms to form a new and improved solution. This is what the authors of (Fu et al. 2013) did by adding selection, crossover and mutation operators to a quantum particle swarm optimization (QPSO). A second approach to hybridization consists of executing different metaheuristics in parallel and allowing cooperation by migrating solutions between the algorithms. This model is often referred to as the island model since each algorithm optimizes its own population. We believe that this second approach is superior since it does not modify the convergence properties of the original algorithms. Cooperative hybrid metaheuristics based on the island model were proposed in (Cadenas et al. 2008), (Li & Yang 2008) and (Izzo et al. 2012). In all three cases, an improvement of the quality of the final solution was observed in multiple

benchmark problems. The publications discussed above all used sequential implementations on CPU. In this paper, we propose a solution that allows hybrid cooperative metaheuristics, but parallelized on GPU in order to significantly accelerate the computation.

4 Recent works

Since the release of CUDA in 2007, research in the field of parallel metaheuristics on GPU has boomed. Many implementations have been published. They all aim to reduce the computation time and each one reveals different advantages and disadvantages. As an example, the authors of (You Zhou & Ying Tan 2009) implement a local variant of the PSO on GPU. Instead of using a global communication scheme as in the original PSO (Kennedy & Eberhart 1995), they restrict the communication of a particle to its two closest neighbors. They achieve a speedup of 11.4x. The authors of (Laguna-Sánchez et al. 2010) implement and compare three different variants of the PSO on GPU, but only parallelize the evaluation of the cost function. They report a maximum speedup of 27x. The authors of (Cárdenas-Montes et al. 2011) discuss the impact of different *thread block* sizes on the performance of the PSO in a GPU implementation. They also limit their parallelization to the evaluation of the cost function. The authors of (Mussi et al. 2011) proposed a GPU-based asynchronous PSO that uses a single CUDA kernel and simultaneously runs independent swarms on different *multiprocessor* blocks. The authors of (Solomon et al. 2011) proposed a similar approach using multiple swarms, but did not limit their implementation to a single kernel and allowed migrations of the particles between the swarms. Finally, the author of (Roberge & Tarbouchi 2012) proposed a parallel implementation on GPU that uses a single population and does not change the behavior of the original PSO. They achieved a speedup of 215x.

To use a metaheuristic as an optimization tool in a software application, one can rely on an existing framework. A framework provides functions or software modules ready to use, but unlike a software library, the framework also defines the generic structure of these modules to allow extensibility. A programmer can therefore use a metaheuristic already implemented in the framework or develop a new metaheuristic following the generic structure imposed by the framework. The framework facilitates the use of metaheuristics and their adaptation to specific problems, but also the improvement or even the development of new metaheuristics. In a study conducted in 2012 (Parejo et al. 2012), the authors identified 33 existing metaheuristic optimization framework (MOF). They selected the top ten and made a comprehensive comparison using 271 evaluation criteria in order to emphasize their strengths, but also identify opportunities for future development. They define the main concepts of a MOF as follows. First, there is the software application that uses a MOF to solve a problem. The MOF provides a set of ready-to-use metaheuristics, but also defines the structure for the development of new metaheuristics. Metaheuristics are independent of the problem and provide a search strategy to iteratively improve the candidate solutions. These solutions are encoded and their decoding is problem specific and necessary before evaluating the solution cost using the objective function. Although metaheuristics are generic, they can use problem specific operators or heuristics to improve the candidate solutions. Using a MOF to solve a specific problem therefore requires the selection of the metaheuristic, the choice of an encoding, the development of a decoding function, the development of a cost function and the development operators or heuristics adapted to the problem and the encoding used.

FOM or *Framework for Metaheuristic Optimization* (Parejo et al. 2003) is one of the first frameworks to be published. This framework uses the object-oriented programming paradigm (OO) and relies on abstract classes to define the structure and clearly separate the problem, the solutions and the metaheuristic, allowing for a better reusability. Two other newer and reputable frameworks are *jMetal* (Durillo & Nebro 2011) and *Opt4J* (Lukasiewycz et al. 2011). Programmed in Java, they also use the OO paradigm to isolate the problem from the optimization method. For parallel execution, both *jMetal* and *Opt4J* offer a limited support by allowing the cost function to be evaluated in parallel on multi-core CPU. For distributed systems, the authors of (Kim et al. 2012) published a framework that initializes the solutions on the different nodes in an island fashion and uses *Opt4J* to optimize independently and in parallel each island. For GPU support, there is currently only one framework available: *ParadisEO-MO-GPU* (Melab et al. 2013). Developed in 2013, this framework is limited to local search algorithms based on a single solution and uses the GPU to generate and evaluate in parallel a large number of neighboring solutions. The cost of the neighboring solutions are then transferred back to the CPU so the algorithm can identify the best neighbor in order to update the current solution. Although the framework provides an acceleration compared to a sequential implementation on CPU, the core of the algorithm remains sequential and runs on the CPU. Moreover, the speedup is limited by the data transfer between the CPU and the GPU, at each iteration of the search.

The major difficulty that limits the extension of current framework for GPU support is their heavy use of the OO paradigm even at the data level. As an example, both *jMetal* and *Opt4J* creates objects for every solutions and every variables within a solution. To access a data element, one must dereference the solution and the variable pointers resulting in two levels of indirect memory addressing. This is acceptable for a sequential implementation on CPU, but not in the case of a parallel implementation on GPU since all these random memory access would not benefit from the large cache line and would be executed sequentially. In the next section, we present *gpuMF*, a framework for parallel metaheuristics on GPU that parallelizes every steps of the optimization algorithm and exploits a high level of data parallelism ensuring maximum speedup.

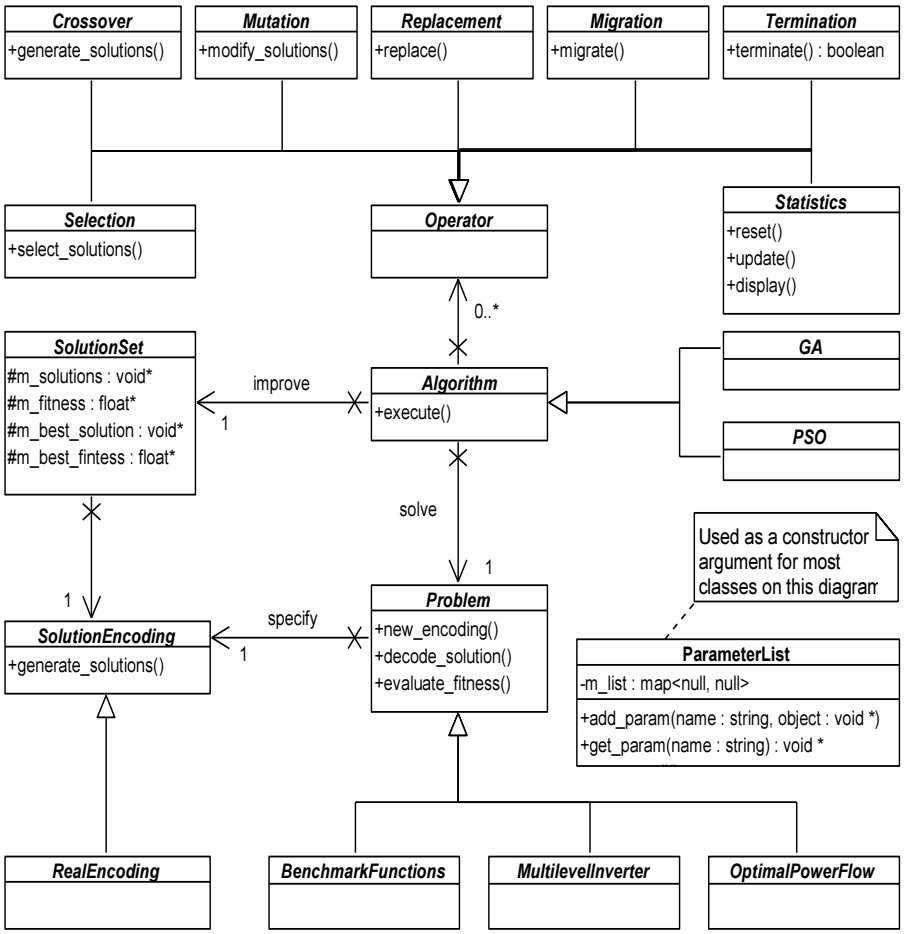
5 *gpuMF*: GPU Metaheuristic Framework

Many papers have been published in recent years on different parallelization strategies for metaheuristics on GPU, but there currently exist not software framework that gather those advancements into a comprehensive, reusable and extensible software tool. To address this shortcoming, we present *gpuMF* (or GPU Metaheuristic Framework), a framework for parallel metaheuristics on GPUs. At a higher level, it relies on the OO paradigm to offers a clear separation between the optimization tool and the problem allowing *gpuMF* to be reused for a wide range of optimization problems. At the lower level, it relies on C arrays to store all the solutions in a continuous fashion allowing coalescent memory access to the GPU global memory. *gpuMF* exploits the intrinsic parallelism found in metaheuristics and fully utilizes the massively parallel architecture of GPUs. Using an XML file, the user can easily select, configure and adapt the metaheuristics for a specific application. Moreover, to provide an optimization tool that is more robust against a wider range of problems, *gpuMF* supports hybrid metaheuristics based on the island model where the candidate solutions are divided into islands and each metaheuristic independently works on its respective island. Collaboration between the different metaheuristics is implemented by a migration process where selected solutions migrate from an island to another following

different migration strategies offering a good balance between the exploration of the search space and convergence towards the final and quasi-optimal solution.

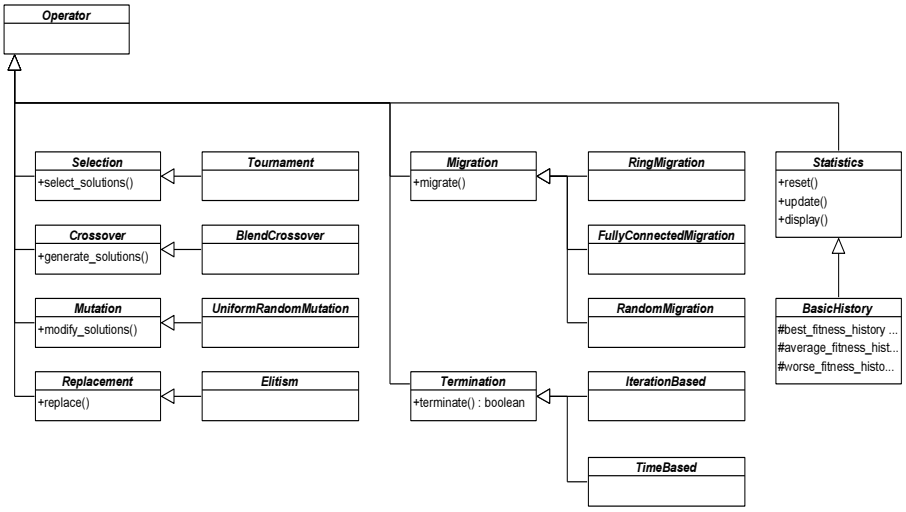
The architecture of the *gpuMF* framework is presented in the various UML diagrams that follow. The highest level class diagram is shown in Figure 4 and defines the core modules that make up the framework and their relationships. The *Algorithm* class uses *Operator's* to modify, improve and optimize solutions to a given problem. Unlike frameworks such at *jMetal* and *Opt4J*, *gpuMF* uses a single class called *SolutionSet* to store all the solutions in a single C array. This ensures that all data is placed sequentially in memory and allows *gpuMF* to exploit data level parallelism, which is essential for a GPU implementation.

Figure 4. UML Class Diagram for all core classes of *gpuMF*



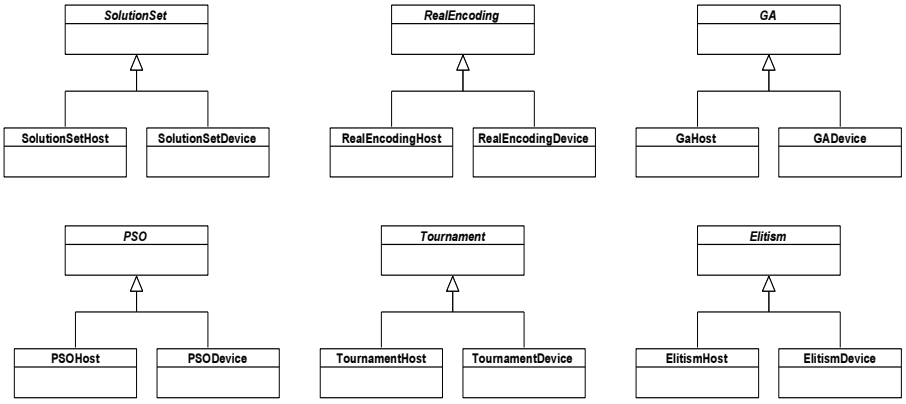
Most of the classes in Figure 4 are abstract (in italics on the diagram). These classes cannot be instantiated, but define the common attributes and methods of the derived classes, those that inherit from the abstract class. We present in Figure 5, the classes derived from *Operator*.

Figure 5. UML Class Diagram for derived classes of the base class *Operator*



Yet, the classes in Figure 5 are also abstract and cannot be instantiated. They define the interfaces (i.e. the functions the user can call) for each of the operators, but do not define their implementation since this latter varies depending if the operator is to be executed on the CPU or the GPU. In fact for every classes in *gpuMF*, there will be a derived class for CPU execution and a derived class for GPU execution. As an example, we present the CPU and GPU implementations for some of the *gpuMF* classes in Figure 6.

Figure 6. UML Class Diagram for derived classes for implementations for CPU and GPU

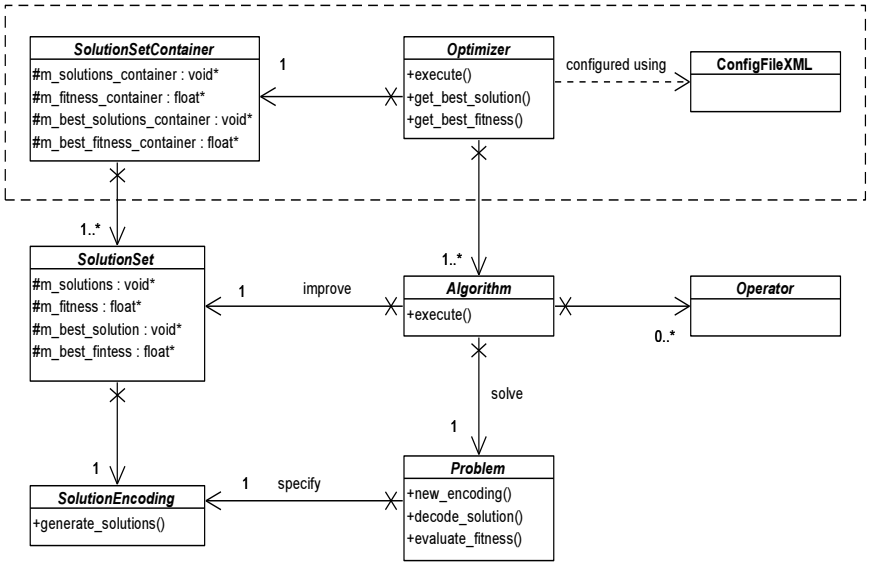


To allow the use of hybrid algorithms composed of different metaheuristics running in parallel that cooperates by exchanging their solutions, we added three classes to the core architecture previously presented in Figure 4. These classes are framed by a dotted line in Figure 7. The *Optimizer* class uses one or more *Algorithm's* to modify and optimize solutions of the *SolutionSetContainer*. The latter class is composed of a *SolutionSet* for each *Algorithm*, but it also ensures that all solutions of the multiple *SolutionSet's* are placed

sequentially in memory. Each *Algorithm* optimizes its own *SolutionSet*, but the true advantage of the *SolutionSetContainer* is that it allows the use of the same mutation operators to exchange solutions within an algorithm (within the *SolutionSet*) or between algorithms (between *SolutionSet's*).

The architecture of *gpuMF* is highly modular and facilitates the reusability of modules between different metaheuristics. However, running an algorithm in *gpuMF* requires the user to programmatically create and assemble each module which can be time consuming and error-prone for a hybrid metaheuristic due to the number and order of modules to be created. To avoid this problem, the *Optimizer* class reads an XML configuration file and automatically creates and assembles all required modules. Our implementation uses the *abstract factory* design pattern (Freeman et al. 2004) to create objects at runtime instead of compile time. The *abstract factory* is derived by two *concrete factories*, one for CPU objects and one for GPU objects. A user can write a single XML configuration file and launch metaheuristics on either the CPU or the GPU. The *Optimizer* module will make sure to use the correct factory when creating objects.

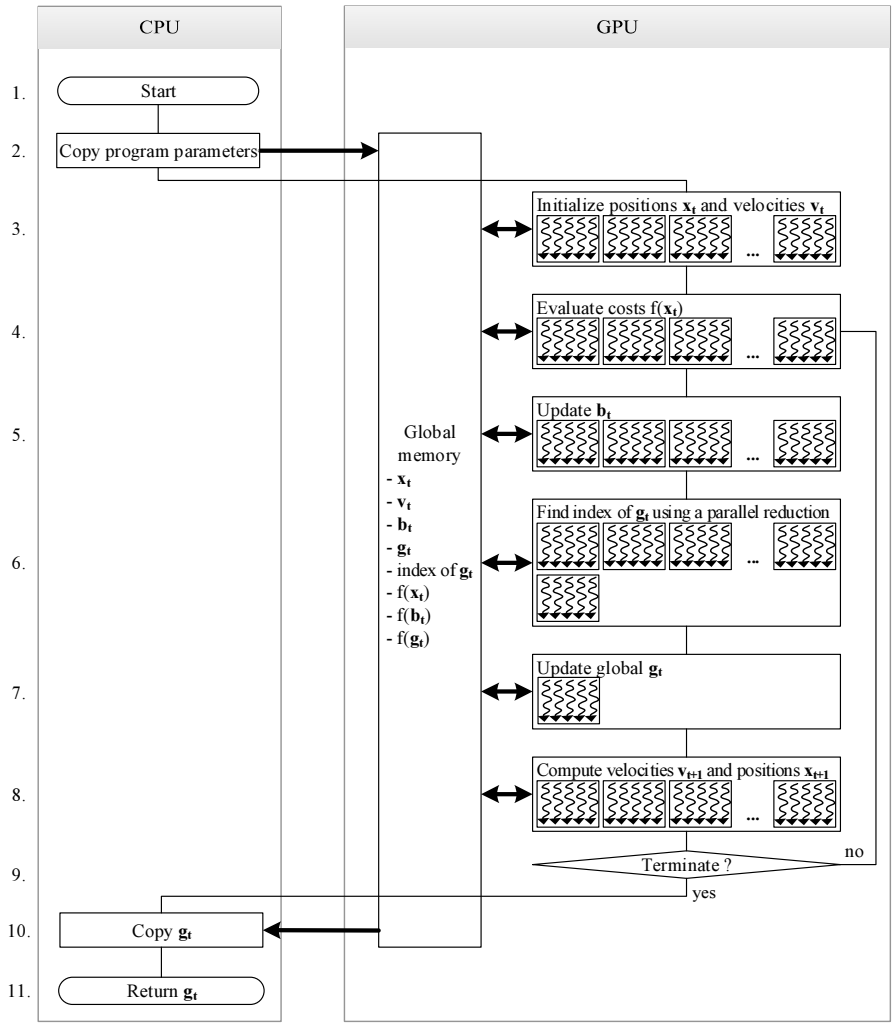
Figure 7. UML Class Diagram for hybrid metaheuristics and XML configuration support



The different figures presented in this section illustrates the structure of the framework, but do not convey the actual behavior of the optimization engine. When the user calls the *execute()* function of the *Optimizer*, the different objects composing the framework will interact at the high level to ensure the proper sequence of execution for the chosen algorithm. However, at the low level, the different objects will call parallel functions or CUDA *kernels* and the actual work or computation will all be done in a massively parallel manner on the GPU. As an example, we show in Figure 8 the flowchart representing the different low level functions that is called by *gpuMF* for the parallel Particle Swarm Optimizer (PSO) algorithm configured with a single island. This figure can be compared to Figure 3 which shows the flowchart for a sequential implementation. In Figure 8, the algorithm is initiated on the CPU, but all the computation is offloaded to the

GPU. The candidate solutions are created, modified and destroyed on the GPU avoiding memory copies over the PCIe bus. Functions at lines 3, 4, 5, 6 and 8 use one thread per candidate solution while the function at line 7 uses one thread per dimension of the global best solutions. To deal with the lack of synchronization between *thread blocks* as imposed by the CUDA programming model, the parallel reduction used at line 6 to find the global best solution is done using two steps or *kernels*. In the first *kernel*, each *thread block* finds the best solutions within its respective data and stores the temporary results in global memory. In the second *kernel*, a single *thread block* is launched and reduces those temporary results to find the global best solution. At the very end, only the final solution is copied back to the CPU and returned to the caller.

Figure 8. Sequence diagram for the parallel PSO



6 Minimization of harmonics in multilevel inverters

Metaheuristics have been used to solve many optimization problems in power systems for which classic methods are inadequate. Yet, despite their effectiveness, their execution time can remain too long. Fortunately, our proposed framework can reduce significantly the execution time and provides an easy way to extend and adapt metaheuristics for a given application. To illustrate this great potential we will show in this section how *gpuMF* can be used to efficiently control multilevel inverters. Multilevel inverters are a popular class of power inverters because of their high voltage operation, their high efficiency, their minimal switching losses and their low electromagnetic interference. These electrical devices use several H-bridges connected in cascade, each powered by a dc voltage source such as in Figure 9. The multilevel inverter control is done by adjusting the commutation angles to produce an ac output by generating a staircase waveform as shown in Figure 10.

The difficulty here is to calculate these angles to produce the desired voltage for a given modulation index while minimizing harmonics which are undesirable.

Figure 9. Schematic of a single-phase cascaded 7-level inverter (or 3-source inverter)

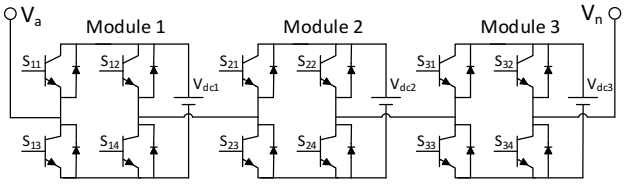
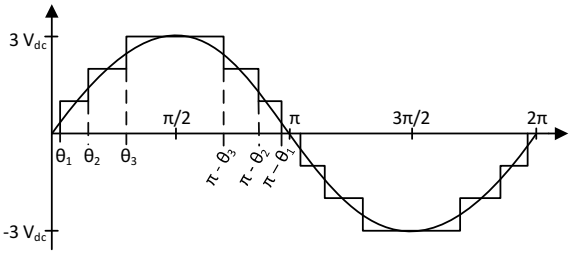


Figure 10. Output voltage waveform for a 7-level inverter



Given a multilevel inverter as shown in Figure 9, the problem of harmonics minimization consists of calculating the optimal switching angles to ensure the magnitude of the fundamental component of the ac output is as close as possible to the desired magnitude for a given modulation index while minimizing the total harmonic distortion (THD). The model presented here is based on (Kaviani et al. 2009), but generalized to consider multilevel inverter with unequal dc sources. For a given modulation index and a series of unequal dc sources, the desired fundamental component amplitude may be calculated as:

$$\mathrm{V}_1^* = \frac{4 * \mathrm{M} * \sum_{s=1}^{S} \mathrm{V}_{dc_s}}{\pi} \tag{3}$$

where S is the number of dc sources and V_{dc_s} is the voltage of the s^{th} source. A Fourier series analysis of the staircase output can be used to determine the amplitude of the fundamental component produced $V_{h=1}$ and the harmonics $V_{h=3,5,7,\ldots}$ as follows:

$$V_h = \frac{4}{h * \pi} \sum_{s=1}^{S} \mathrm{V}_{dc_s} \cos(h * \theta_s) \tag{4}$$

where h is the order of the harmonic and θ_s is the s^{th} switching angle. These angles should satisfy the following constraint to ensure a stepped output:

$$0 \leq \theta_1 \leq \cdots \leq \theta_s \leq \cdots \leq \theta_S \leq \frac{\pi}{2} \tag{5}$$

Finally, the THD is calculated by dividing the square root of the sum of the harmonics by the fundamental component as follows:

$$\text{THD} = \frac{\sqrt{\sum_{h=3,5,7,9,\ldots} V_h^2}}{V_1} \tag{6}$$

Several solutions have been successfully used to solve the problem of minimizing harmonics of a multilevel inverter. The authors of (Yu Liu et al. 2009) proposed a deterministic approach relying on the Newton-Raphson method to control an inverter with three sources. Their approach is effective for small problems, but is not scalable to inverters with a large number of sources. For this reason, recent research focuses mainly on non-deterministic methods such as artificial neural networks (ANN) and metaheuristics. In (Filho et al. 2011), a two-layer perceptron network is used to control an inverter with four sources. Again, the scalability remains to be verified.

Yet, the use of inverters with a large number of dc sources offers interesting advantages. These inverters generate a better waveform, a smaller THD and require a simpler filtering. A typical application is the generation of ac voltage from a series of solar panels. For these applications, metaheuristics such as the particle swarm optimization (PSO) (Kaviani et al. 2009), the genetic algorithm (GA) (Yousefpoor et al. 2012) or the bee algorithm (BA) (Kavousi et al. 2012) have been used successfully. Based on the above equations, to solve the problem, one can use the following cost function as we previously proposed in (Roberge et al. 2014).

$$f_{cost}(\vec{\theta}) = \left(1{,}000 * \frac{\mathrm{V}_1^* - \mathrm{V}_1}{\mathrm{V}_1^*}\right)^4 + \text{THD} \tag{7}$$

In this equation, the first term penalizes severely any solution $\vec{\theta}$ for which the error between V_1 and V_1^* is greater than 0.1%. The second term penalizes solutions with a high THD. The advantage of using metaheuristics is that these non-deterministic algorithms remain effective even when the dimension of the problem grows. However, their execution time is not negligible and can vary from 1.2 s to 77 s depending on the number of dc sources and the number of harmonics considered in the minimization of the THD (Roberge et al. 2014). These values are too large to allow real-time control. Yet, one can easily notice that the calculation of the cost function has a large data-level parallelism. More particularly, the THD in equation (6) is the sum of the harmonics and each harmonic in equation (4) is a sum of terms. It is therefore possible to calculate each term in parallel and use a parallel reduction operation to compute the THD. The calculation of the cost function can then be accelerated by a parallel execution on a GPU. Moreover, by using *gpuMF*, the entire optimization algorithm can execute on a GPU allowing for a higher frequency of control inputs even for inverters with many dc sources while minimizing several harmonics.

Our implementation of the cost function uses one *thread block* per solution and starts with a bitonic parallel sort (Peters et al. 2010) in order to ensure all candidate solutions respect the constraint at equation (5). It then uses one thread per harmonics and proceed to the computation in parallel. Finally, a parallel reduction sums the harmonics to compute the THD. To configure *gpuMF*, we use a parallel PSO with a population of 512 candidate solutions divided into 16 islands of 32 each. The inertia factor ω, the personal influence factor c_1 and the social influence factor c_2 used in equations (1) are respectively set to 0.7298, 1.496 and 1.496 as suggested in (Clerc & Kennedy 2002). The termination criterion is based on the number of iterations completed and is set to 400. A migrations following a bi-directional ring topology is used and exchanges the 8 best solutions every 50 iterations (so 7 times during the optimization).

To reflect the good functioning of our approach, we optimized and show in Figure 11, the switching angles for a 10-source inverter (V_{DC1}=1.00, V_{DC2}=0.88, V_{DC3}=0.85,

V_{DC4}=1.05, V_{DC5}=1.08, V_{DC6}=1.12, V_{DC7}=0.92, V_{DC8}=1.10, V_{DC9}=1.05, V_{DC10}=0.95 p.u.) for modulation index M between 0.5 and 0.95, while considering the first 100 harmonics. The THD and the error between the desired V_1^* and the generated V_1 are plotted in Figures 12 and 13. The results confirms that the cost function previously defined is appropriate and guaranties extremely low errors of V_1 while successfully minimizing the THD up to the first 100 harmonics. Compared to the results published in (Yousefpoor et al. 2012), (Kaviani et al. 2009) and (Kavousi et al. 2012), the method proposed in this paper provides a much lower THD due to the higher number of dc sources used which would not have been possible using a sequential implementation due to the long execution time. To illustrate this fact, we also plotted in Figures 12 and 13, the THD and the error between the desired V_1^* and the generated V_1 for inverters with 5, 10, 15 and 20 dc sources. It is obvious that using a higher number of DC sources is advantageous and leads to a reduced THD.

Figure 11. Switching angles found by the parallel PSO implemented in *gpuMF* for a 10-source inverter versus the modulation index M

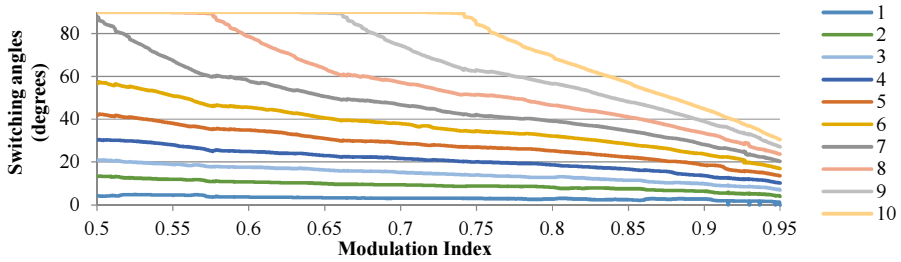


Figure 12. Voltage THD versus the modulation index M

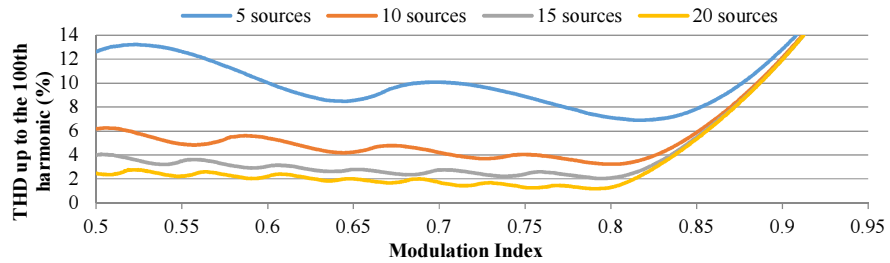
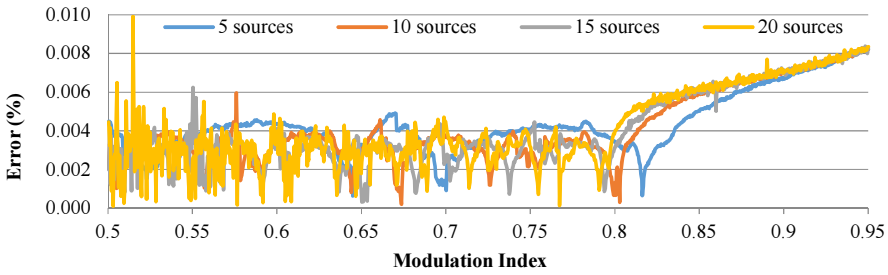


Figure 13. Error between the fundamental component of the output and the desired fundamental component versus the modulation index M



In a second test, to demonstrate the correctness of the proposed method, the modulation index was fixed to 0.8 and the optimal angles for the 10-source inverter described above were computed. The source voltage and the switching angles calculated by the PSO are listed in Table 1. The stepped output is shown in Figure 14. This stepped waveform was analyzed in MATLAB® using a Fast Fourier transform (FFT) and the harmonics computed are listed in Table 2 and plotted in Figure 15. The values obtained with the FTT are exactly equal to the ones computed using equation (4) which demonstrates the correctness of the mathematical model used. The THD was measured to 3.24% which is smaller than the THD of 12.89% and 12.52% respectively achieved in (Yu Liu et al. 2009) and (Kavousi et al. 2012), also using a modulation index of 0.8. This difference is due to the higher number of dc sources allowed by the parallel approach proposed here.

Table 1. Optimal switching angles (rad) found by our parallel PSO in gpuMF for a 10-source inverter and a modulation index of 0.8

Source	Voltage (p.u.)	Switching angle (rad)	Source	Voltage (p.u.)	Switching angle (rad)
V_{dc1}	1.00	0.0486	V_{dc6}	1.12	0.5613
V_{dc2}	0.88	0.1437	V_{dc7}	0.92	0.6827
V_{dc3}	0.85	0.2239	V_{dc8}	1.10	0.8121
V_{dc4}	1.05	0.3259	V_{dc9}	1.05	0.9872
V_{dc5}	1.08	0.4405	V_{dc10}	0.95	1.2114

Figure 14. Output voltage using the switching angles found by the parallel PSO implemented in *gpuMF* for a 10-source inverter and a modulation index of 0.8

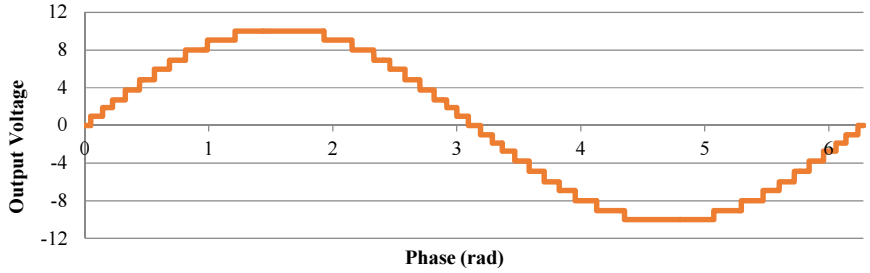
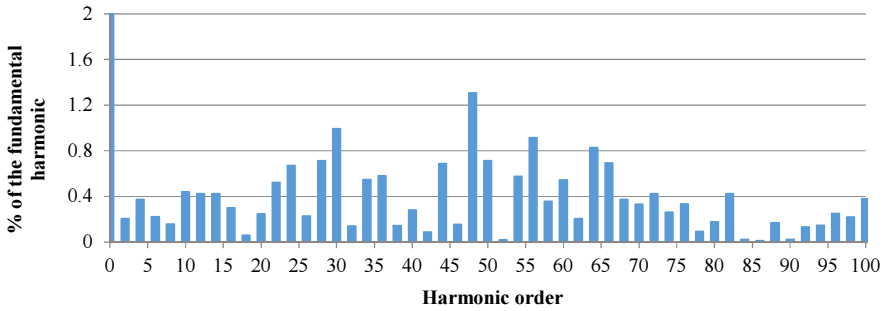


Table 2. Harmonics amplitude (in % of the fundamental component) of the output voltage shown in Figure 14.

Order	Amp.	Order	Amp.	Order	Amp.	Order	Amp.	Order	Amp.
3rd	0.2021	23rd	0.5240	43rd	0.0871	63rd	0.2023	83rd	0.4216
5th	0.3734	25th	0.6711	45th	0.6855	65th	0.8277	85th	0.0215
7th	0.2209	27th	0.2265	47th	0.1526	67th	0.6924	87th	0.0117
9th	0.1550	29th	0.7141	49th	1.3066	69th	0.3723	89th	0.1701
11th	0.4398	31st	0.9920	51st	0.7139	71st	0.3295	91st	0.0239
13th	0.4232	33rd	0.1405	53rd	0.0173	73rd	0.4239	93rd	0.1322
15th	0.4230	35th	0.5472	55th	0.5753	75th	0.2605	95th	0.1455
17th	0.2985	37th	0.5820	57th	0.9129	77th	0.3334	97th	0.2508
19th	0.0577	39th	0.1427	59th	0.3575	79th	0.0925	99th	0.2203
21st	0.2469	41st	0.2788	61st	0.5447	81st	0.1778	101st	0.3777

Figure 15. Harmonics amplitude (in % of the fundamental component) of the output voltage shown in Figure 14. Although not entirely visible, the left most bar represents the fundamental component and goes up to 100%.



In a final test, to demonstrate the performance improvement provided by our proposed parallel framework for metaheuristics on GPU, we generated two versions of the optimization program, one for a sequential execution on the CPU and a second one for a parallel execution on the GPU. The sequential version was run on an Intel Xeon E3 1230 CPU with a frequency of 3.6 GHz. The parallel version was run on an EVGA NVIDIA GTX750 Ti SC with 640 cores running at 1,255 MHz. We then measured the execution time and computed the speedup for inverters with different numbers of sources while considering various numbers of harmonics in the minimization. The results in Figures 16 and 17 show sequential executions times ranging from 223 ms to 40.0 s and reduced parallel execution times from 42.5 ms to 140.9 ms which represents speedups from 5.25x to 276.7x. This truly demonstrate the advantage of our parallel framework.

Figure 16. Execution times of the sequential PSO on GPU in *gpuMF* and the parallel PSO on GPU in *gpuMF* to optimize inverters with different number of dc sources and harmonics

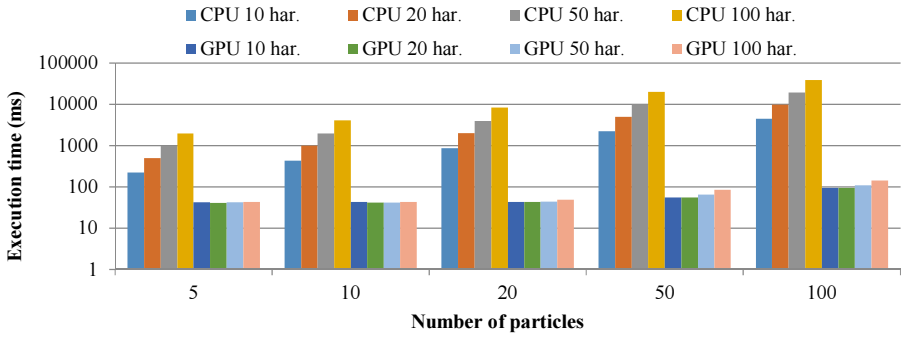
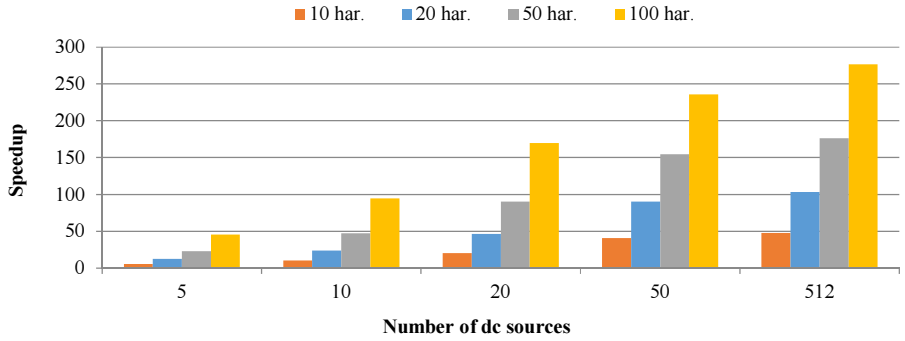


Figure 17. Speedup of the sequential PSO on GPU in *gpuMF* and the parallel PSO on GPU in *gpuMF* to optimize inverters with different number of dc sources and harmonics



7 Conclusion

In this paper, we presented *gpuMF*, a novel framework for parallel hybrid cooperative metaheuristics on GPU. Our framework successfully addresses two drawbacks of metaheuristics. Firstly, metaheuristics are general purpose optimization algorithms and are very efficient against a wide range of applications. However, they require significant computing power and their execution time can be too long for time critical applications. *gpuMF* addresses this limitation by an efficient parallelization on the GPU using CUDA. Compared to other frameworks, ours allows every steps of the algorithm to be parallelized and executed on the GPU maximizing the speedup achieved. Secondly, each metaheuristic demonstrates specific strengths and their effectiveness varies from one problem to another. To improve their robustness, *gpuMF* is designed to easily allow cooperative hybrid metaheuristics. Finally, to demonstrate the efficiency of the proposed framework, we successfully used *gpuMF* to minimize harmonics in multilevel inverters. Due to its parallelization on GPU, *gpuMF* was able to optimize inverters with a large number of sources while considering many harmonics.

References

Cadenas, J.M., Garrido, M.C. & Muñoz, E., 2008. A Hybrid System of Nature Inspired Metaheuristics. In N. Krasnogor et al., eds. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*. Studies in Computational Intelligence. Springer Berlin Heidelberg, pp. 95–104. Available at: http://dx.doi.org/10.1007/978-3-540-78987-1_9.

Cárdenas-Montes, M. et al., 2011. Effect of the block occupancy in GPGPU over the performance of particle swarm algorithm. In *Proceedings of the 10th international conference on Adaptive and natural computing algorithms - Volume Part I*. Ljubljana, Slovenia: Springer-Verlag, pp. 310–319.

Clerc, M. & Kennedy, J., 2002. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1), pp.58–73.

Durillo, J.J. & Nebro, A.J., 2011. jMetal: A Java framework for multi-objective optimization. *Adv. Eng. Softw.*, 42(10), pp.760–771.

Filho, F. et al., 2011. Real-Time Selective Harmonic Minimization for Multilevel Inverters Connected to Solar Panels Using Artificial Neural Network Angle Generation. *Industry Applications, IEEE Transactions on*, 47(5), pp.2117–2124.

Freeman, E. et al., 2004. *Head First Design Patterns*, O' Reilly \& Associates, Inc. Available at: http://portal.acm.org/citation.cfm?id=1076324.

Fu, Y. et al., 2013. Route Planning for Unmanned Aerial Vehicle (UAV) on the Sea Using Hybrid Differential Evolution and Quantum-Behaved Particle Swarm Optimization. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, PP(99), pp.1–1.

Izzo, D., Ruciński, M. & Biscani, F., 2012. The Generalized Island Model. In F. Fernández de Vega, J. I. Hidalgo Pérez, & J. Lanchares, eds. *Parallel Architectures and Bioinspired Algorithms*. Studies in Computational Intelligence. Springer Berlin Heidelberg, pp. 151–169. Available at: http://dx.doi.org/10.1007/978-3-642-28789-3_7.

Kaviani, A.K. et al., 2009. PSO, an effective tool for harmonics elimination and optimization in multi-level inverters. In *2009 4th IEEE Conference on Industrial Electronics and Applications, 25-27 May 2009*. 2009 4th IEEE Conference on Industrial Electronics and Applications. IEEE, pp. 2902–7.

Kavousi, A. et al., 2012. Application of the Bee Algorithm for Selective Harmonic Elimination Strategy in Multilevel Inverters. *IEEE Transactions on Power Electronics*, 27(4), pp.1689–96.

Kennedy, J. & Eberhart, R., 1995. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks. Part 4 (of 6), Nov 27 - Dec 1 1995*. IEEE International Conference on Neural Networks - Conference Proceedings. IEEE, pp. 1942–1948.

Kim, J. et al., 2012. A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing. *Journal of Parallel and Distributed Computing*, 72(4), pp.564–578.

Laguna-Sánchez, G.A. et al., 2010. Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU. *Journal of Applied Research and Technology*, 7(3), pp.292–309.

Li, C. & Yang, S., 2008. An Island Based Hybrid Evolutionary Algorithm for Optimization. In X. Li et al., eds. *Simulated Evolution and Learning*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 180–189. Available at: http://dx.doi.org/10.1007/978-3-540-89694-4_19.

Lukasiewycz, M. et al., 2011. Opt4J: a modular framework for meta-heuristic optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland: ACM, pp. 1723–1730.

Melab, N. et al., 2013. ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. Amsterdam, The Netherlands: ACM, pp. 1189–1196.

Melab, N., Cahon, S. & Talbi, E.-G., 2006. Grid computing for parallel bioinspired algorithms. *Special Issue: Parallel Bioinspired Algorithms Special Issue: Parallel Bioinspired Algorithms*, 66(8), pp.1052–1061.

Mussi, L., Nashed, Y.S.G. & Cagnoni, S., 2011. GPU-based asynchronous particle swarm optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland: ACM, pp. 1555–1562.

Parejo, J. et al., 2012. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3), pp.527–561.

Parejo, J.A. et al., 2003. FOM: A Framework for Metaheuristic Optimization. In P. A. Sloot et al., eds. *Computational Science — ICCS 2003*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 886–895. Available at: http://dx.doi.org/10.1007/3-540-44864-0_91.

Peters, H., Schulz-Hildebrandt, O. & Luttenberger, N., 2010. Fast in-place sorting with CUDA based on bitonic sort. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*. Wroclaw, Poland: Springer-Verlag, pp. 403–410.

Ragnarsson, R.M., Stefánsson, H. & Ásgeirsson, E.I., 2011. Meta-Heuristics in Multi-Core Environments. *Engineering and Risk Management*, 1(0), pp.457–464.

Rauber, T. & Rünger, G., 2010. *Parallel Programming: for Multicore and Cluster Systems*, Springer. Available at: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/364204817X.

Roberge, V. & Tarbouchi, M., 2012. Parallel Particle Swarm Optimization on Graphical Processing Unit for Pose Estimation. *World Scientific and Engineering Academy and Society Transactions on Computers*, 11(6), pp.170–179.

Roberge, V., tarbouchi, M. & Okou, A., 2014. Strategies to Accelerate Harmonic Minimization in Multilevel Inverters using a Parallel Genetic Algorithm on Graphical Processing Unit. *Power Electronics, IEEE Transactions on*, PP(99), pp.1–1.

Sena, G.A., Megherbi, D. & Isern, G., 2001. Implementation of a parallel Genetic Algorithm on a cluster of workstations: Traveling Salesman Problem, a case study. *Workshop on Bio-inspired Solutions to Parallel Computing problems*, 17(4), pp.477–488.

Solomon, S., Thulasiraman, P. & Thulasiram, R., 2011. Collaborative multi-swarm PSO for task matching using graphics processing units. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. Dublin, Ireland: ACM, pp. 1563–1570.

Talbi, E.-G., 2009. *Metaheuristics: From Design to Implementation (Wiley Series on Parallel and Distributed Computing)*, Wiley. Available at: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0470278587.

Weyland, D., Montemanni, R. & Gambardella, L.M., 2013. A metaheuristic framework for stochastic combinatorial optimization problems based on GPGPU with a case study on the probabilistic traveling salesman problem with deadlines. *Metaheuristics on GPUs*, 73(1), pp.74–85.

Wolpert, D.H. & Macready, W.G., 1997. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1), pp.67–82.

Yousefpoor, N. et al., 2012. THD Minimization Applied Directly on the Line-to-Line Voltage of Multilevel Inverters. *Industrial Electronics, IEEE Transactions on*, 59(1), pp.373–380.

You Zhou & Ying Tan, 2009. GPU-based parallel particle swarm optimization. *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pp.1493–1500.

Yu Liu, Hoon Hong & Huang, A.Q., 2009. Real-Time Calculation of Switching Angles Minimizing THD for Multilevel Inverters With Step Modulation. *Industrial Electronics, IEEE Transactions on*, 56(2), pp.285–293.