View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Parallel Hybrid Evolutionary Algorithms on GPU

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi

To cite this version:

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi. Parallel Hybrid Evolutionary Algorithms on GPU. IEEE Congress on Evolutionary Computation (CEC), 2010, Barcelone, Spain. inria-00520466

HAL Id: inria-00520466

https://hal.inria.fr/inria-00520466

Submitted on 23 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Parallel Hybrid Evolutionary Algorithms on GPU

Thé Van Luong, *Member, IEEE Student*, Nouredine Melab, *Member, IEEE* El-Ghazali Talbi

***Abstract*— Over the last years, interest in hybrid metaheuristics has risen considerably in the field of optimization. Combinations of methods such as evolutionary algorithms and local searches have provided very powerful search algorithms. However, due to their complexity, the computational time of the solution search exploration remains exorbitant when large problem instances are to be solved. Therefore, the use of GPU-based parallel computing is required as a complementary way to speed up the search. This paper presents a new methodology to design and implement efficiently and effectively hybrid evolutionary algorithms on GPU accelerators. The methodology enables efficient mappings of the explored search space onto the GPU memory hierarchy. The experimental results show that the approach is very efficient especially for large problem instances.**

I. INTRODUCTION

In combinatorial optimization, metaheuristics allow to iteratively solve in a reasonable time NP-hard complex problems. According to the number of solutions handled at each iteration, two main categories of metaheuristics are often distinguished: evolutionary algorithms (EAs) and local searches (LSs). EAs are population-oriented as they manage a whole population of individuals, what confers them a good exploration power. Indeed, they allow to explore a large number of promising regions in the search space. On the contrary, LSs work with a single solution which is iteratively improved by exploring its neighborhood in the solution space. Therefore, they are characterized by better local intensification capabilities.

Theoretical and experimental studies have shown that the hybridization between EAs and LSs improves the effectiveness (quality of provided solutions) and the robustness of the metaheuristics [1]. Nevertheless, as it is generally CPU time-consuming it is not often fully exploited in practice. Indeed, experiments with hybrid EAs are often stopped without convergence being reached. That is the reason why, in designing hybrid EAs, there is often a compromise between the number of individuals to use and the computational complexity to explore it. As a consequence, in such algorithms, there is often a reduction of the size of the explored space at the expense of the effectiveness. To deal with such issues, only the use of parallelism allows to design efficient hybrid EAs.

Nowadays, GPU computing is recognized as a powerful way to achieve high-performance on long-running scientific applications [2]. With the arrival of the general-purpose computation on graphics processings units (GPGPU) paradigm, EAs on GPU have generated a growing interest. Many works on GPU have been proposed: genetic algorithm [3], genetic programming [4], and evolutionary programming [5].

Parallel hybrid EAs for solving real-world problems are good challenges for GPU computing. However, to the best of our knowledge very few research works have been investigated on hybrid EAs on GPU. Unlike the parallel evaluation of the population on GPU for traditional EAs, the parallel evaluation of the neighborhood on GPU for LSs is not straightforward. Indeed, several scientific challenges mainly related to the hierarchical memory management have to be faced. The major issues are the efficient distribution of data processing between CPU and GPU, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc.

The main objective of this paper is to deal with such issues for the re-design of hybrid EAs to allow solving of large scale optimization problems on GPU architectures. We propose a new general methodology for building efficient hybrid EAs on GPU. This methodology is based on a three-level decomposition of the GPU hierarchy allowing a clear separation between generic and problem-dependent hybrid evolutionary features. Several challenges are dealt with: (1) the distribution of the search process among the CPU and the GPU minimizing the data transfer between them; (2) finding the efficient mapping of the neighborhood of the currently processed solution to GPU threads. (3) using efficiently the coalescing and texture memory in the context of hybrid EAs.

The remainder of the paper is organized as follows. On the one hand, Section II highlights the principles of hybrid EAs. On the other hand, for a better understanding of the difficulties of using the GPU architecture, its characteristics are described according to the three-level decomposition. Section III provides a depth look on the three-level decomposition. First, generic concepts for designing parallel hybrid EAs on GPU are presented (high-level). Second, efficient mappings between state-of-the-art optimization structures and GPU threads are performed (intermediate-level). Then, the memory management on GPU adapted to hybrid EAs is depicted (low-level). To validate the approaches presented in this paper, Section IV reports the performance results obtained for the quadratic assignment problem (QAP). Finally, a discussion and some conclusions of this work are drawn in Section V.

II. PARALLEL HYBRID EVOLUTIONARY ALGORITHMS ON GPU

A. Hybrid Evolutionary Algorithms

Two completing goals govern the design of a metaheuristic: exploration and exploitation. Exploration is needed to ensure that every part of the space is searched enough to

T.V. Luong, N. Melab and E-G. Talbi are from both INRIA Lille - Nord Europe and CNRS/LIFL Labs, Université de Lille1, France (e-mail: The-Van.Luong@inria.fr; Nouredine.Melab@lifl.fr; El-Ghazali.Talbi@lifl.fr)

provide a reliable estimate of the global optimum. Exploitation is important since the refinement of the current solution will often produce a better solution.

EAs are powerful in the exploration of the search space and weak in the exploitation of the solutions found whereas LS algorithms such as tabu search are powerful optimization methods in terms of exploitation. The two classes of algorithms have complementary strengths and weaknesses. The LSs will try to optimize locally, while the EAs will try to optimize globally. In this paper, we focus on hybrid EAs in which a LS is embedded into an EA. This class of hybrid algorithms is very popular and has been applied successfully to many optimization problems [6].

In these hybrid metaheuristics, instead of using a blind operator acting regardless of the fitness of the original individual and the operated one, a heuristic operator considers an individual as the origin of its search applies itself, and finally replaces the original individual by the enhanced one. The evolutionary operators replaced or extended are generally mutation, crossover or initialization.

In general, evaluating a fitness function for each solution is frequently the most costly operation of metaheuristics. That is the reason why, for hybrid EAs, executing the iterative process of a LS on large neighborhoods requires a large amount of computational resources. Consequently, parallelism arises naturally when dealing with a neighborhood, since each of the solutions belonging to it is an independent unit. Due to this, the performance of LS algorithms and thus of hybrid EAS is particularly improved when running in parallel. Parallel design and implementation of metaheuristics have been studied as well on different architectures [7], [8].

B. GPU Hierarchy Decomposition

Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, this architecture provides tremendous computational horsepower and very high memory bandwidth compared to traditional CPUs. Since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computation. A complete review of GPU architecture can be found in [2].

The adaptation of hybrid EAs on GPU requires to take into account at the same time the characteristics and underlined issues of the GPU architecture and the metaheuristics parallel models. Since the evaluation of the neighborhood is generally the time-consuming part of hybrid EAs, we focus on the re-design of LS algorithms on GPU. In this section, we propose a three-level decomposition of the GPU adapted to the popular parallel iteration-level model [1] (generation and evaluation of the neighborhood in parallel) allowing a clear separation of the GPU memory hierarchical management concepts (Fig. 1). The different aspects of this model will be discussed throughout this section.

In the high-level layer, task distribution is clearly defined: the CPU manages the whole sequential hybrid evolutionary process and the GPU is dedicated to the parallel evaluation of solutions at the other levels. The intermediate-level layer focuses on the generation and partitioning of the LS neighborhood on GPU. Afterwards, GPU memory management of the evaluation function computation is done at low-level.

1) High-level Layer - General GPU Model: This level describes common GPU concepts which are language-independent. In general-purpose computing on graphics processing units, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of the GPU during the execution of programs. In optimization problems, the types of data which are manipulated are the data inputs of the tackled problem and the solution representation.

Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple autonomous processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device simultaneously by several processors in parallel. Regarding the iteration-level parallel model, generation and evaluation of neighboring candidates are done in parallel. Therefore, according to the Master-Worker paradigm, these two steps can be executed in parallel on GPU. In other words, a kernel is associated with the generation and evaluation of the neighborhood.

Memory transfer from the CPU to the device memory is a synchronous operation which is time consuming. In the case of LS methods, memory copying operations from CPU to GPU are essentially the solution duplication operations which generate the neighborhood. Afterwards, the kernel representing the generation and evaluation of the neighborhood is processed at both intermediate-level and low-level. Regarding transfers from GPU to CPU, the results of the evaluation function (fitnesses) of each candidate solution of the neighborhood are stored in an array structure.

2) Intermediate-level Layer - Threads Mapping: The intermediate-level layer focuses on the neighborhood generation on GPU. This kernel handling is dependent of the general-purpose language. For instance, CUDA or OpenCL are parallel computing environments which provide an application programming interface. These toolkits introduce a model of threads which provides an easy abstraction for single-instruction and multiple-data (SIMD) architecture. A thread on GPU can be seen as an element of the data to be processed. Compared to CPU threads, GPU threads are lightweight. That means that changing the context between two threads is not a costly operation. Therefore, GPU threads management is clearly identified as the main task of the generation step of LS neighborhood.

Regarding their spatial organization, threads are organized within so called thread blocks. A kernel is executed by multiple equally threaded blocks. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar

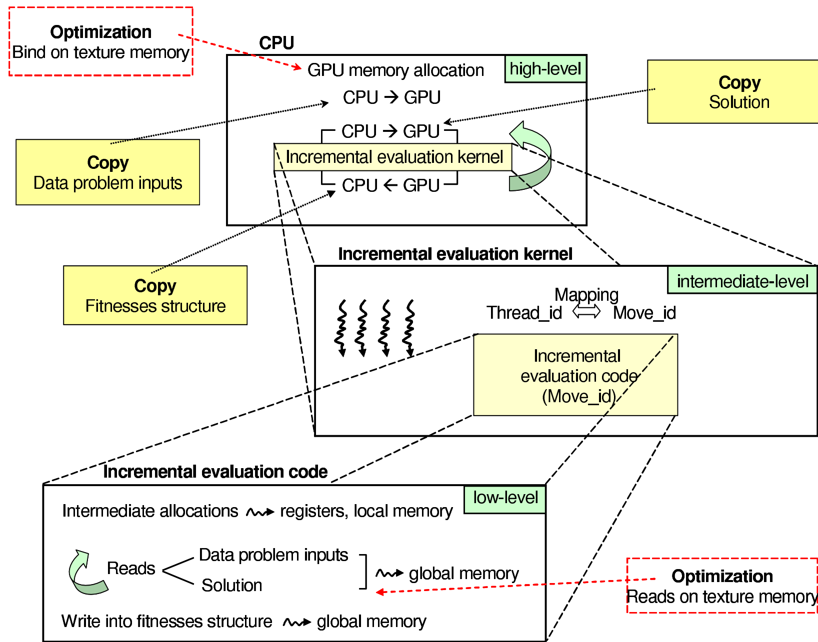

Fig. 1. The three-level decomposition of the GPU hierarchy in accordance with the hybrid evolutionary process.

way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors. Thus, a unique *id* can be deduced for each thread to perform computation on different data. Regarding LS algorithms, a move which represents a particular neighbor candidate solution can also be associated with a unique *id*. However, according to the solution representation of the problem, finding a corresponding *id* for each move is not straightforward.

3) Low-level Layer - Kernel Memory Management: The low-level layer focuses on the specific part of the evaluation function. As stated before, each GPU thread executes the same kernel i.e. each candidate solution of the neighborhood executes the same evaluation function code. From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors are used according to the SPMD model, threads share the same code and have access to different memory areas.

Communication between the CPU host and its device is done through the global memory. For hybrid EAs, more exactly for the evaluation function, the global memory stores the data input of problems and their solution representation. Since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories. Graphics cards provide also read-only texture memory to accelerate operations such as 2D or 3D mapping. Texture memory units are provided to allow faster graphic operations. In the case of hybrid EAs, binding texture on global memory can provide an alternative optimization. Registers among streaming processors are partitioned among the threads running on it, they constitute fast access memory. In the evaluation function kernel code, each declared variable is automatically put into registers. Local memory is a memory abstraction and is not an actual hardware component. In fact, local memory resides in the global memory allocated by the compiler. Complex structures such as declared array will reside in local memory.

The memory management in the low-level layer is problem-specific. A clear understanding of the characteristics described above is required to provide an efficient implementation of the evaluation function. According to the SPMD model, the same code is executed by all the neighbors in parallel and the resulting fitnesses must be stored into the fitnesses structure (global memory) previously mentioned.

III. Design and Implementation of Hybrid Evolutionary Algorithms on GPU

In this section, the focus is on the design and the implementation of hybrid EAs according to the three-level decomposition model presented in the previous section.

A. The Proposed Scheme of Parallelization

In the high-level layer, the CPU sends the number of expected running threads to the GPU, then candidate neighbors are generated and evaluated on GPU (at intermediate-level and low-level), and finally newly evaluated solutions are returned back to the host. This model can be seen as a cooperative model between the CPU and the GPU. Indeed, the GPU is used as a coprocessor in a synchronous manner. The resource-consuming part i.e. the generation and evaluation kernel, is calculated by the GPU and the rest is handled by the CPU. Adapting traditional hybrid EAs to GPU is not a straightforward task because hierarchical memory

management on GPU has to be handled. As previously said, memory transfers from CPU to GPU are slow and these copying operations have to be minimized. We propose (see Fig. 2) a methodology to adapt hybrid EAs on GPU in a generic way.

First of all, at initialization stage, memory allocations on GPU are made: data inputs and candidate solution of the problem must be allocated. Since GPUs require massive computations with predictable memory accesses, a structure has to be allocated for storing all the neighborhood fitnesses at different addresses. Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of incremental evaluation. Second, problem data inputs, initial candidate solution and additional structures associated to this solution have to be copied on the GPU. It is important to notice that problem data inputs are a read-only structure and never change during all the execution of hybrid EAs. Therefore, their associated memory is copied only once during all the execution. Third, comes the parallel iteration-level, in which each neighboring solution is generated (intermediate-level), evaluated (low-level) and copied into the neighborhood fitnesses structure. Fourth, since the order in which candidate neighbors are evaluated is undefined, the neighborhood fitnesses structure has to be copied to the host CPU. Finally, a specific LS solution selection strategy is applied to this structure: the exploration of the neighborhood fitnesses structure is done by the CPU in a sequential way. The process is repeated until a stopping criterion is satisfied.

B. *Efficient Mapping of the Neighborhood Structure on GPU*

In this subsection, a focus is made on the neighborhood generation in the intermediate-level layer. As suggested in Fig. 3, the challenging issue of the intermediate-level layer is to find efficient mappings between a thread *id* and a particular neighbor.

1) Binary Representation: In binary representations, a solution is coded as a vector (string) of bits. The neighborhood representation for binary problems is based on the Hamming distance. The neighborhood of a given solution is obtained by flipping one bit of the solution (for a Hamming distance of one).

A mapping between LS neighborhood encoding and GPU threads is quiet trivial. Indeed, on the one hand, for a binary vector of size n, the size of the neighborhood is exactly n. On the other hand, threads are provided with a unique *id*. This way, the kernel associated with the incremental evaluation is launched with n threads (each neighbor is associated with a single thread), and the size of the neighborhood fitnesses structure allocated on GPU is n. As a result, a $\mathbb{N} \to \mathbb{N}$ mapping is straightforward.

2) Discrete Vector Representation: Discrete vector representation is an extension of binary encoding using a given alphabet Σ. In this representation, each variable takes its value over the alphabet Σ. Assume that the cardinality of the alphabet Σ is k, the size of the neighborhood is $(k-1) \times n$ for a discret vector of size n.

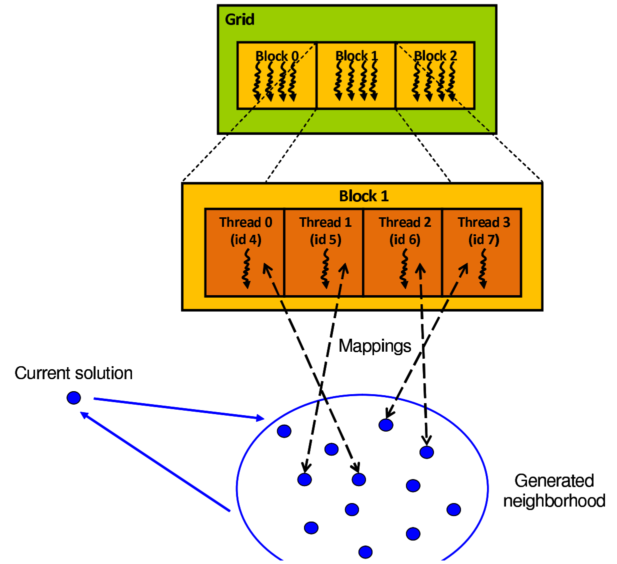

Fig. 3. Mappings between threads and neighbors

Let id be the identity of the thread corresponding to a given candidate solution of the neighborhood. Compared to the initial solution which has allowed to generate the neighborhood, $id/(k-1)$ represents the position which differs from the initial solution and $id\%(k-1)$ is the available value from the ordered alphabet Σ (both using zero-index based numbering).

As a consequence, a $\mathbb{N} \to \mathbb{N}$ mapping is possible. $(k-1) \times n$ threads execute the incremental evaluation kernel, and a neighborhood fitnesses structure of size $(k-1) \times n$ has to be provided.

3) Permutation Representation: Building a neighborhood by pair-wise exchange operations (known as the 2-exchange neighborhood) is a standard way for permutation problems. For a permutation of size n, the size of the neighborhood is $\frac{n \times (n-1)}{2}$.

The mapping here is a generalization of the mapping for traditional permutation problems. For a single permutation encoding a mapping between a neighbor is composed by two element indexes and threads are identified by a unique *id*. As a result, a $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$ mapping has to be considered to transform one index into two ones.

The incremental evaluation kernel is executed by $\frac{n \times (n-1)}{2}$ threads, and the size of neighborhood fitnesses structure is $\frac{n \times (n-1)}{2}$. Unlike the previous representations, for permutation encoding a mapping between a neighbor and a GPU thread is not straightforward. Indeed, on the one hand, a neighbor is composed by two element indexes (a swap in a permutation). On the other hand, threads are identified by a unique *id*. As a result, a $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$ mapping has to be considered to transform one index into two ones. In a similar way, a $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ mapping is required to transform two indexes into one. Finding a nearly constant time mapping is clearly a challenging issue for permutation representation.

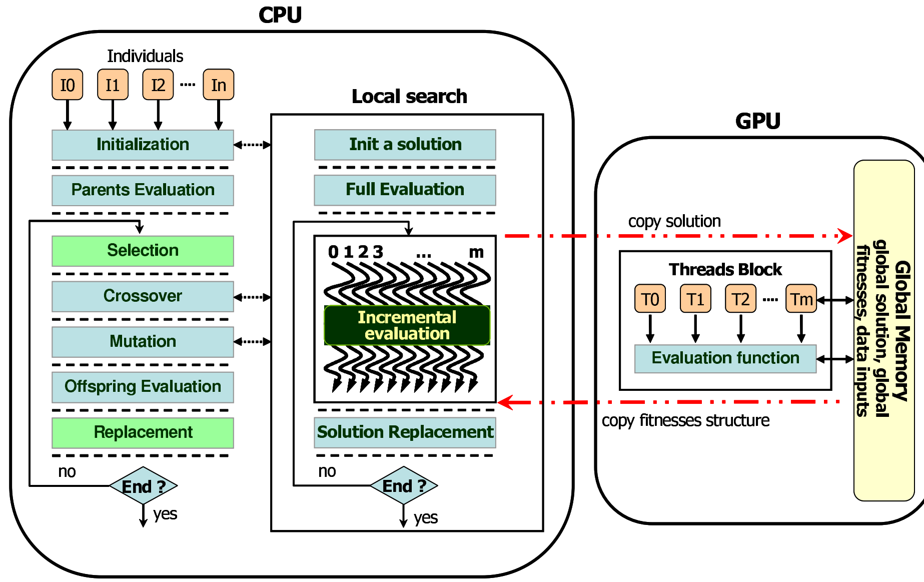

Fig. 2. In the high-level layer, the CPU manages the whole hybrid evolutionary process. The generation and the evaluation of the LS neighborhood are performed on parallel on GPU.

Proposition 1: **Two-to-one index transformation**

Given i and j the indexes of two elements to be exchanged in the permutation representation, the corresponding index $f(i,j)$ in the neighborhood representation is equal to $i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2}$, where n is the permutation size.

Proposition 2: **One-to-two index transformation**

Given $f(i,j)$ the index of the element in the neighborhood representation, the corresponding index i is equal to $n - 2 - \lfloor \frac{\sqrt{8 \times (m - f(i,j) - 1) + 1} - 1}{2} \rfloor$ and j is equal to $f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1$ in the permutation representation, where n is the permutation size and m the neighborhood size.

The proofs of these transformations can be found in [9].

C. Memory Management on GPU

In this subsection, the focus is on the memory management in the low-level layer. Understanding the GPU memory organization and issues is useful to provide an efficient implementation of parallel metaheuristics.

1) Memory Coalescing Issues: Each block of GPU threads is split into SIMD groups of threads called *warps*. At any clock cycle, each processor of the multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data. Global memory is conceptually organized into a sequence of 128-byte segments. The number of memory transactions performed for a half-warp will be the number of segments having the same addresses than those used by that half-warp. Fig. 4 illustrates an example of the low-level layer for a simple vector addition.

For more efficiency, global memory accesses must be coalesced, which means that a memory request performed by consecutive threads in a half-warp is associated with precisely one segment. The requirement is that threads of the same warp must read global memory in an ordered pattern. If per-thread memory accesses for a single half-warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. In the example of vector addition, memory accesses to the vectors a and b are fully coalesced, since threads with consecutive thread indices access contiguous words.

Otherwise, accessing scattered locations results in memory divergence and requires the processor to perform one memory transaction per thread. The performance penalty for non-coalesced memory accesses varies according to the size of the data structure. Regarding LS evaluation kernels, coalescing is difficult when global memory access has a data-dependent unstructured pattern (especially for permutation representation). As a result, non-coalesced memory accesses imply many memory transactions and it can lead to a significant performance decrease for hybrid EAs.

2) Texture Memory: Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. The use of texture memory is a solution for reducing memory transactions due to non-coalesced accesses. Texture memory provides a surprising aggregation of capabilities including the ability to cache global memory. Indeed, texture memory provides an alternative memory access path that can be bound to regions of the global memory. Each texture unit has some internal memory that buffers data from global memory. Therefore, texture memory can be seen as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements do not apply to texture memory accesses. The use of texture memory is well adapted for hybrid EAs for the following reasons:

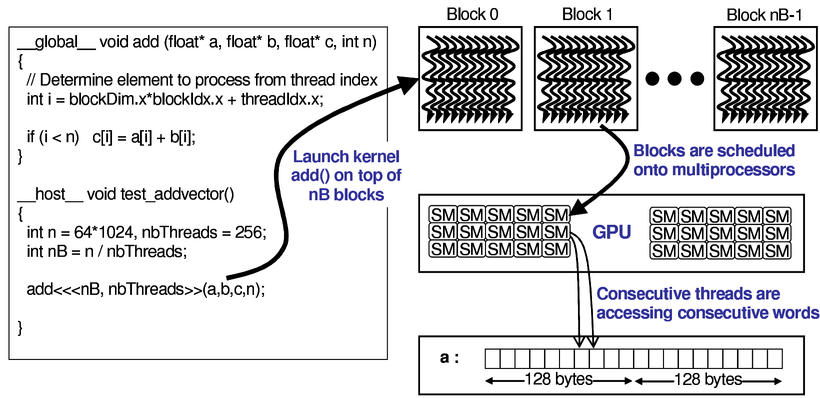

Fig. 4. An example of kernel execution for vector addition.

- Matrices accesses are frequent in the computation of incremental evaluation methods. Then, using texture memory can provide a high performance improvement by reducing the number of memory transactions.
- Texture memory is a read-only memory i.e. no writing operations can be performed on it. This memory is adapted to hybrid EAs since the matrices and the solution representation are also read-only values.
- Minimizing the number of times that data goes through cache can increase significantly the efficiency of algorithms. In hybrid EAs, the matrices and the solution representation do not often require a large amount of allocated space memory. As a consequence, these structures can take advantage of the 8KB cache per multiprocessor of texture units.
- Cached texture data is laid out to give best performance for 1D/2D access patterns. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. Since the problem inputs of hybrid EAs are often 2D matrices and 1D solution vectors, these structures can be bound to texture memory.

IV. Experimentation

To validate the approaches presented in this paper, the QAP has been implemented on GPU. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices of positive integers. Finding a solution of the QAP is equivalent to finding a permutation $\pi = (1, 2, \ldots, n)$ that minimizes the objective function:

$$z(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)}$$

A. Configuration

For this problem, a hybrid EA with an iterative local search (ILS) has been implemented on GPU with CUDA. The embedded method is a tabu search (TS) and the perturbation applied in the ILS process uses a random number of pair-wise exchanges. The number of ILS iterations has been fixed to 3 and the number of TS iterations to 10000. The chosen neighborhood is based on a Hamming distance of 3 where and its associated size is equal to $\frac{n\times(n-1)\times(n-2)}{6}$ where n is the instance size. Generating a neighbor is obtained by performing two swaps from the initial solution. The tabu list size has been set to $2 \times \lfloor\sqrt{\frac{n\times(n-1)\times(n-2)}{6}}\rfloor$.

Regarding the EA, first, random initializations are performed with a population of 10 individuals. Second, the selection operator is a deterministic tournament with a size of 2. Third, the crossover selects the common attributes in both parents and the remaining entries are chosen at random (crossover rate fixed to 100%). Fourth comes the hybridization where the mutation operator is replaced by the ILS-TS (mutation rate fixed to 100%). Finally, the replacement strategy is a $(\mu+\lambda)$ replacement and the number of generations has been fixed to 10.

The problem has been implemented using a permutation representation. The incremental evaluation function has a time complexity of $O(n)$, and the number of created threads is equal to $\frac{n\times(n-1)\times(n-2)}{6}$. The used configuration for the experiments is a Core 2 Duo 2.67Ghz with a NVIDIA GTX 280 card (30 multiprocessors).

B. Measures in Terms of Efficiency and Effectiveness

For each instance, a standalone mono-core CPU implementation, a CPU-GPU, and a CPU-GPU version using texture memory are considered. The average time has been measured for 30 runs. Average values of the evaluation function have been collected and the number of successful tries (hits) is also represented. The associated standard deviation for each average measurement is shown in sub-index. Since the computational time is too exorbitant for the tai80a and tai100a instances, the average expected time for the CPU implementation has been deduced from the base of two executions. Table I reports the obtained results for the hybrid EA.

Generate and evaluate the neighborhood in parallel on GPU provides an efficient way to speed-up the search process in comparison with a single CPU. Indeed, from the instance tai30a, the GPU version is already faster than the CPU one (acceleration factor of $\times 5.2$). As long as the problem size increases, the speed-up grows significantly (up to $\times 7.2$ for the tai100a instance).

TABLE I

HYBRID EVOLUTIONARY ALGORITHM FOR DIFFERENT QAP INSTANCES.

Instance	Best known value	Hits	CPU time	GPU time	Acceleration	GPUTexture time	Acceleration
tai30a	1818146	27/30	1h 15min	14min 25s	×5.2	8min 50s	×8.5
tai35a	2422002	23/30	2h 24min	25 min 36s	×5.6	12min 56s	×11.1
tai40a	3139370	18/30	3h 54min	39min	×5.9	18min 16s	×12.8
tai50a	4938796	10/30	10h 2min	1h 37min	×6.2	45 min	×13.2
tai60a	7205962	6/30	20h 17min	3h 9min	×6.4	1h 30min	×13.4
tai80a	13511780	4/30	*66h*	9h 48min	×6.7	4h 45min	×13.8
tai100a	21052466	2/30	*177h*	24h 29min	×7.2	12h 6min	×14.6

Due to high misaligned accesses to global memories (flows and distances in QAP), non-coalescing memory reduces the performance of the GPU implementation. Binding texture on global memory allows to overcome the problem. Indeed, from the instance tai30a, using texture memory starts providing significant acceleration factor of $\times 8.5$. GPU keeps accelerating the hybrid evolutionary process as long as the size grows (up to $\times 14.6$).

Regarding the quality of solutions, in comparison with the literature [6], the obtained results by the proposed hybrid EA is quiet competitive. Indeed, Taillard instances larger than 30 are well-known for their difficulty and the proposed algorithm is able to find the best known value with a significant rate success for most instances.

The conclusion from this experiment indicates that the use of GPU provides an efficient way to deal with large neighborhoods. Indeed, since the LS neighborhood is based on a Hamming distance of 3 (two swaps), the proposed hybrid EA is unpractical in terms of single CPU computational resources for large instances such as tai80a or tai100a (more than 60h per run). So, implementing this algorithm on GPU has allowed to exploit parallelism in such neighborhood to improve the robustness/quality of provided solutions.

C. Discussion and Performance of the Proposed Approach

It is well-known that CPU/GPU communication might be a major bottleneck in the performance of GPU applications. In our proposed iteration-level parallel model (generation and evaluation of the neighborhood on GPU), the data transfers are essentially the fitnesses which are copied from the GPU to the CPU at each iteration of the LS process ($\frac{n\times(n-1)\times(n-2)}{6}$ fitnesses). At first sight, this number of copying operations may seem important. Therefore, a natural way of thinking would be to move the entire LS process on GPU. This way, it would drastically reduce the data transfers between the GPU and the CPU.

However, this different approach raises several issues because the threads execution on GPU works in an asynchronous manner and the threads synchronization is only local to each threads block. Consequently, in the context of a fully distributed scheme on GPU, the global synchronization of GPU threads to find the best admissible neighbor is not straightforward without a loss of performance.

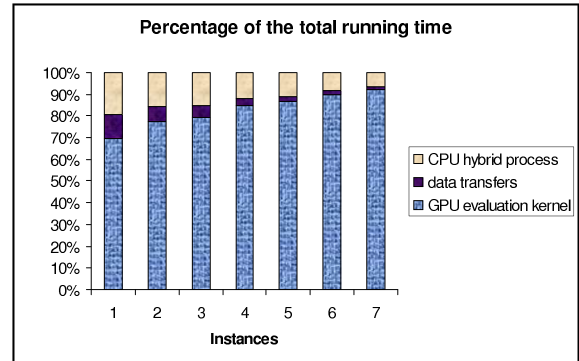

Fig. 5. Average percentage of the time spent by each operation in the hybrid EA on GPU. The taillard instances are ordered according to their size.

A more important point concerns the time spent on data transfers relative to the total execution time of the hybrid EA. Indeed, if this time was negligible in comparison with the total runnning time, then the interest to fully distribute the LS process on GPU would become limited. To go on this idea, we propose to make an analysis of the percentage of the time spent by each major operation in our GPU-based implementation to evaluate the impact in terms of efficiency (see Fig. 5).

A first observation that can be made is about the data transfers between the CPU and the GPU. The time associated with the transfers keeps decreasing with the problem size increase. Indeed, from the first instance (tai30a), this time corresponds to 11% of the total running time and it reaches the value of 1% for the last instance (tai100a). As a consequence, the time dedicated to the data transfers in the iteration-level parallel model is not significant in comparison with the LS evaluation process.

Another observation concerns the time spent by the generation and the evaluation of the neighborhood on GPU (evaluation kernel) which represents most of the total running time. For instance, for the fourth instance (tai50a), the time associated with the evaluation of the neighborhood corresponds to 85% of the total execution time. This time grows accordingly with the instance size (around 90% for tai60a, tai80a and tai100a).

According to the two previous observations, in the case of

hybrid EAs, the advantage of a fully distributed scheme of the LS process on GPU over our approach may be limited in terms of efficiency.

Regarding the time spent by the hybrid EA process, only the execution time of the LS process without the evaluation of the neighborhood is reported. Indeed, the time related to the evolutionary process is insignificant since it has been measured at less than one second for each instance. Thus, there is no interest to distribute the evolutionary process on GPU. Regarding the hybrid evolutionary process including the LS selection (CPU hybrid process), it varies from 19% to 7% of the total running time. One might think that a full distribution of the LS process on GPU might improve the performance. However, as previously said, the establishment of a fully distributed scheme of the LS process on GPU is not natural in accordance with the LS design. Furthermore, even it was feasible without a loss of performance, the benefits in terms of speed-up would not be really significant.

V. Discussion and Conclusion

Hybrid EAs having complementary behaviors allow to improve the effectiveness and robustness in combinatorial optimization. Their exploitation for solving real-world problems is possible only by using a great computational power. High-performance computing based on the use of computational GPUs is recently revealed as an efficient way to use the huge amount of resources at disposal. However, the exploitation of parallel models is not trivial and many issues related to the GPU memory hierarchical management of this architecture have to be considered. To the best of our best of knowledge, GPU-based parallel hybrid EAs approaches have never been proposed.

In this paper, the idea of our approach is based on a three-level decomposition of the GPU hierarchy. This way, efficient mapping of the iteration-level parallel model on the hierarchical GPU is proposed. In the high-level layer, the CPU manages the whole hybrid evolutionary process and let the GPU be used as a coprocessor dedicated to intensive calculations. The goal of the intermediate-level layer is to find efficient mappings between neighborhood candidate solutions and GPU threads in nearly-constant time. Memory management is handled at the low-level layer where optimization based on texture memory is applied to the evaluation function kernel.

The designed and implemented approaches have been experimentally validated on the QAP. The experiments indicate that GPU computing allows not only to speed up the search process, but also to exploit large neighborhoods structures to improve the quality of the obtained solutions. For instance, hybrid EAs based on a Hamming distance of three is unpractical on traditional machines because of their high computational cost. So, GPU computing has permitted their achievement and the obtained results are particularly promising in terms of effectiveness. In this paper, we have also investigated on how a full distribution of the LS process on GPU would not improve significantly the performance in terms of speed-up in comparison with the proposed approach.

Beyond the improvement of the effectiveness, the parallelism of GPUs allows to push far the limits in terms of computational resources. One possible extension of this work is to investigate a multi-GPU approach at the iteration-level to speed up the neighborhood evaluation of the individual being processed. Indeed, multi-GPU architectures offer an opportunity to perform independent computations in parallel on each separate card. Therefore, the main idea to accelerate the hybrid evolutionary search process is to perform one LS per GPU in parallel. To achieve this in an efficient way, a mixed multi-core and GPU approach must be considered. This way, each CPU core executes in parallel one individual (LS algorithm) according to the previously designed three-level decomposition.

A next perspective will be to integrate the GPU-based re-design of metaheuristics in the ParadisEO platform [10]. This framework has been developed for the reusable and flexible design of parallel hybrid metaheuristics dedicated to the mono and multiobjective optimization. ParadisEO is based on a clear conceptual separation of metaheuristics concepts, and can be seen as a white-box object-oriented with reusable components. The Parallel Evolving Objects (PEO) module of ParadisEO includes the well-known parallel and distributed models for metaheuristics. This module will be extended with multi-core and GPU-based generic components.

References

[1] E.-G. Talbi, *Metaheuristics: From design to implementation*. Wiley, 2009.

[2] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, “Program optimization carving for gpu computing,” *J. Parallel Distribributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.

[3] S. Tsutsui and N. Fujimoto, “Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study,” in *GECCO ’09*. New York, NY, USA: ACM, 2009, pp. 2523–2530.

[4] W. Banzhaf, S. Harding, W. B. Langdon, and G. Wilson, “Accelerating genetic programming through graphics processing units.” in *Genetic Programming Theory and Practice VI*, 2009, pp. 1–19.

[5] T.-T. Wong and M. L. Wong, “Parallel evolutionary algorithms on consumer-level graphics processing unit,” in *Parallel Evolutionary Computations*, 2006, pp. 133–155.

[6] A. Misevicius, “A fast hybrid genetic algorithm for the quadratic assignment problem,” in *GECCO*, M. Cattolico, Ed. ACM, 2006, pp. 1257–1264.

[7] E. Alba, E.-G. Talbi, G. Luque, and N. Melab, *Parallel Metaheuristics: A New Class of Algorithms*, ser. Wiley Series on Parallel and Distributed Computing. Wiley, 2005, ch. 4. Metaheuristics and Parallelism, pp. 79–104.

[8] N. Melab, S. Cahon, and E.-G. Talbi, “Grid computing for parallel bioinspired algorithms,” *J. Parallel Distributed Computing*, vol. 66, no. 8, pp. 1052–1061, 2006.

[9] T. V. Luong, N. Melab, and E.-G. Talbi, “Parallel Local Search on GPU,” INRIA, Research Report RR-6915, 2009. [Online]. Available: http://hal.inria.fr/inria-00380624/en/

[10] S. Cahon, N. Melab, and E.-G. Talbi, “Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics,” *J. Heuristics*, vol. 10, no. 3, pp. 357–380, 2004.