# Local Search Algorithms on Graphics Processing Units. A Case Study: the Permutation Perceptron Problem

Th Van Luong, Nouredine Melab, and El-Ghazali Talbi

INRIA Dolphin Project / Opac LIFL CNRS
40 avenue Halley, 59650 Villeneuve d'Ascq Cedex FRANCE.
The-Van.Luong@inria.fr, [Nouredine.Melab, El-Ghazali.Talbi]@lifl.fr

**Abstract.** Optimization problems are more and more complex and their resource requirements are ever increasing. Although metaheuristics allow to significantly reduce the computational complexity of the search process, the latter remains time-consuming for many problems in diverse domains of application. As a result, the use of GPU has been recently revealed as an efficient way to speed up the search. In this paper, we provide a new methodology to design and implement efficiently local search methods on GPU. The work has been experimented on the permuted perceptron problem and the experimental results show that the approach is very efficient especially for large problem instances.

**Key words:** GPU-based metaheuristics, local search algorithms on GPU.

## 1 Introduction

Nowadays, optimization problems become increasingly large and complex, forcing the use of parallel computing for their efficient and effective resolution. Indeed, although near-optimal algorithms such as local search (LS) methods allow to reduce the temporal complexity of their resolution, they are unsatisfactory to tackle large problems. Therefore, parallel computing has recently undergone a significant evolution with the emergence of new high performance computing environments including accelerators such as GPUs.

Recently, the use of graphics processors has been extended to general application domains such as computational science [1]. Indeed, GPUs are very efficient at manipulating computer graphics, and their parallel structure makes them more efficient than general-purpose CPUs for a range of complex algorithms. This is why it would be very interesting to exploit this huge capacity of computing to implement parallel metaheuristics. However, there only exists few research works related to evolutionary algorithms on GPU [2–4]. Indeed, the design and implementation of parallel optimization methods raise several issues related to the characteristics of these methods and those of the new hardware execution environments at the same time.

Several scientific challenges mainly related to the hierarchical memory management on GPU have to be considered: 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 parallel LS models to allow solving of large scale optimization problems on GPU architectures. We propose a new general methodology for building efficient parallel LS methods on GPU. This methodology is based on a three-level decomposition of the GPU hierarchy allowing a clear separation between generic and problem-dependent LS features.

To be validated the work has been experimented on the permuted perceptron problem (PPP) introduced by Pointcheval [5]. The problem is a cryptographic identification scheme based on NP-complete problems, which seems to be well suited for resource constrained devices such as smart cards. The proposed work has been experimented using three GPU configurations with different performance capabilities in terms of threads that can be created simultaneously.

The remainder of the paper is organized as follows: In Section 2, the characteristics of the GPU architecture are described according to the three-level decomposition. Section 3 presents generic concepts for designing parallel LS methods on GPU (high-level). In Section 4, efficient mappings between state-of-the-art LS structures and NVIDIA CUDA model are performed (intermediate-level). A depth look on memory management in CUDA adapted to LS heuristics is depicted in Section 5 (low-level). Section 6 reports the performance results obtained for the PPP mentioned above. Finally, a discussion and some conclusions of this work are drawn in Section 7.

## 2 Graphics Processing Units and Three-level Decomposition

Driven by the demand for high-definition 3D graphics, GPUs have evolved into a highly parallel, multithreaded and manycore environment. 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 [6].

The adaptation of LS algorithms on GPU requires to take into account at the same time the characteristics and underlined issues of the GPU architecture and the LS parallel models. In this section, we propose a three-level decomposition of the GPU adapted to the popular parallel iteration-level model [7] (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 the three-level decomposition model will be discussed throughout the next sections.

In the high-level layer, task distribution is clearly defined: the CPU manages the whole sequential LS process and the GPU is dedicated to the parallel evaluation of solutions at the other levels. The intermediate-level layer focuses on

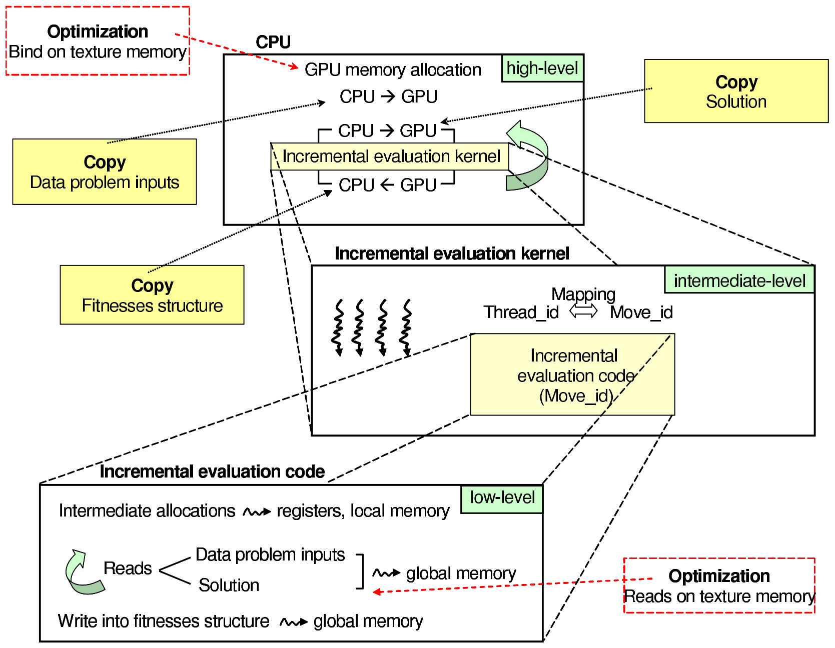


**Fig. 1.** Three-level decomposition

the generation and partitioning of the LS neighborhood on GPU. Afterwards, GPU memory management of the evaluation function computation is done at low-level.

## 2.1 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 GPU during the execution of programs. In LS algorithms, 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, a kernel on GPU can be associated with these two steps.

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.

### 2.2 Intermediate-level Layer: CUDA Threading Model

The intermediate-level layer focuses on the neighborhood generation on GPU. This kernel handling is dependent of the general-purpose language. CUDA was chosen because the toolkit introduces a model of threads which provides an easy abstraction for SIMD architecture [8]. A thread on GPU can be seen as an element of the data to be processed and 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 way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor. 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.

### 2.3 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. From a hardware point of view, 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 LS algorithms, 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). Graphics cards provide also read-only texture memory to accelerate operations such as 2D mapping. In the case of LS algorithms, 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. 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.

## 3 Design of Parallel Local Search Algorithms on GPU

In this section, the focus is on the re-design of the iteration-level parallel model. This model fits well with the high-level layer since parallel LS concepts are generic. Designing parallel LS model is a great challenge as nowadays there is no generic GPU-based LS algorithms to the best of our knowledge.

### 3.1 A General Model for LS Algorithms

According to the SPMD model, multiple autonomous processors simultaneously execute the same program at independent points. Therefore, the mapping at the high-level layer between the LS iteration-level parallel model and the GPU model becomes quiet natural.

First, 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 where 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.

### 3.2 The Proposed GPU-based Algorithm

Adapting traditional LS methods to GPU is not a straightforward task because hierarchical memory management on GPU has to be handled. We propose (see algorithm 1) a methodology to adapt LS methods 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 (lines 4 and 5). Since GPUs require massive computations with predictable memory accesses, a structure has to be allocated for storing all the neighborhood fitnesses at different addresses (line 6). Additional solution structures which are problem-dependent can also be allocated (line 7). Second, all the allocated structures have to be copied on the GPU (lines 8 to 10). Since problem data inputs are a read-only structure, 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

**Algorithm 1** Local Search Template on GPU

```
1:  Choose an initial solution
2:  Evaluate the solution
3:  Specific LS initializations
4:  Allocate problem data inputs on GPU device memory
5:  Allocate a solution on GPU device memory
6:  Allocate a neighborhood fitnesses structure on GPU device memory
7:  Allocate additional solution structures on GPU device memory
8:  Copy problem data inputs on GPU device memory
9:  Copy the solution on GPU device memory
10: Copy additional solution structures on GPU device memory
11: repeat
12:     for each generated neighbor in parallel on GPU do
13:         Incremental evaluation of the candidate solution
14:         Insert the resulting fitness into the neighborhood fitnesses structure
15:     end for
16:     Copy neighborhood fitnesses structure on CPU host memory
17:     Specific LS solution selection strategy on the neighborhood fitnesses structure
18:     Specific LS post-treatment
19:     Copy the chosen solution on GPU device memory
20:     Copy additional solution structures on GPU device memory
21: until a stopping criterion satisfied
```

the neighborhood fitnesses structure (from lines 12 to 15). Fourth, since the order in which candidate neighbors are evaluated is undefined, the neighborhood fitnesses structure has to be copied to the host CPU (line 16). Then a specific LS solution selection strategy is applied to this structure (line 17) on CPU. Finally, after a new candidate has been selected, this latter and its additional structures are copied to the GPU (lines 19 and 20). The process is repeated until a stopping criterion is satisfied.

## 4 Efficient Mappings of Local Search Structures on GPU

The neighborhood structures play a crucial role in the performance of LS methods and are problem-dependent. In this section, a focus is made on the neighborhood generation in the intermediate-level layer.

The challenging issue of this level is to find efficient mappings between a thread *id* and a particular neighbor. Indeed, on the one hand, the thread *id* is represented by a single index. On the other hand, the move representation of a neighbor varies according to the neighborhood. In the following, we provide a methodology to deal with different structures of the literature.

### 4.1 Binary Representation

In binary representation, a solution is coded as a vector of bits. The neighborhood representation for binary problems is based on the Hamming distance where 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*. That way, the kernel associated to the generation and evaluation steps is launched with $n$ threads (each neighbor is associated to a single thread), and the size of the neighborhood fitnesses structure allocated on GPU is $n$. As a result, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping is straightforward.

## 4.2 Discrete Vector Representation

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

Let *id* be the identity of the thread corresponding to a given candidate solution of the neighborhood. Compared to the initial solution which 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 $\Sigma$ (both using zero-index based numbering).

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

## 4.3 Permutation Representation

Building a neighborhood by pairwise exchange operations is a standard way for permutation problems. For a permutation of size $n$, the size of the neighborhood is $\frac{n \times (n-1)}{2}$.

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 $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping has to be considered to transform one index into two ones. In a similar way, a $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping is required to transform two indexes into one.

**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.*

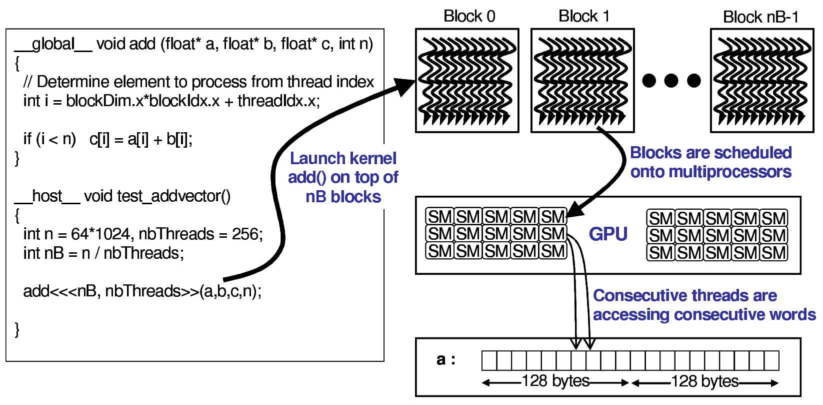
The proofs of these two index transformations can be found in [9]. The generation and evaluation kernel is executed by $\frac{n \times (n-1)}{2}$ threads, and the size of the neighborhood fitnesses structure is $\frac{n \times (n-1)}{2}$. Notice that for binary problem encodings, the mapping of a neighborhood based on a Hamming distance of two can be done in a similar manner.

## 5 Memory Management of Local Search Algorithms on GPU

Task repartition between CPU and GPU and efficient thread mappings in parallel LS heuristics have been proposed on both high-level and intermediate-level layers. In this section, 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 LS heuristics.

### 5.1 Memory Coalescing Issues

In CUDA, each block of 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. 2 illustrates an example of the low-level layer for a simple vector addition.



```
__global__ void add (float* a, float* b, float* c, int n)
{
  // Determine element to process from thread index
  int i = blockDim.x*blockIdx.x + threadIdx.x;

  if (i < n)   c[i] = a[i] + b[i];
}

__host__ void test_addvector()
{
  int n = 64*1024, nbThreads = 256;
  int nB = n / nbThreads;

  add<<<nB, nbThreads>>>(a,b,c,n);

}
```

Block 0
Block 1
Block nB-1
Launch kernel add() on top of nB blocks
Blocks are scheduled onto multiprocessors
SM
GPU
Consecutive threads are accessing consecutive words
a :
128 bytes
128 bytes

**Fig. 2.** An example of kernel execution for 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. As a result, non-coalesced memory accesses imply many memory transactions and it can lead to a significant performance decrease for LS methods.

Notice that in the latest cards (200-series), due to the relaxation of the coalescing rules, applications developed in CUDA get better global memory performance.

## 5.2 Texture Memory

Optimizing the performance of CUDA applications often involves optimizing data accesses which includes the appropriate use of the various CUDA 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, 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 LS algorithms for the following reasons:

- Data accesses are frequent in the computation of LS 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 LS algorithms since the problem data and the solution representation are also read-only values.
- Minimizing the number of times that data goes through cache can increase the efficiency of algorithms. In most of optimization problems, problem inputs 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 optimization problem inputs are generally 2D matrices or 1D solution vectors, LS structures can be bound to texture memory.

## 6 Application to the Permuted Perceptron Problem

An $\epsilon$-vector is a vector with all entries being either +1 or -1. Similarly an $\epsilon$-matrix is a matrix in which all entries are either +1 or -1. The PPP is defined as follows according to [5]:

**Definition 1.** *Given an $\epsilon$-matrix $A$ of size $m \times n$ and a multiset $S$ of non-negative integers of size $m$, find an $\epsilon$-vector $V$ of size $n$ such that $\{\{(AV)_j / j = \{1, \ldots, m\}\}\} = S$.*

As the iteration-level parallel model does not change the semantics of the sequential algorithm, the effectiveness in terms of quality of solutions is not addressed here. Only execution times and acceleration factors are reported. The objective is to evaluate the impact of a GPU-based implementation in terms of efficiency.

A generic tabu search has been implemented on GPU using a binary encoding. The adaptation to GPU of the tabu search is straightforward according to the proposed GPU algorithm in the high-level layer (see Algorithm 1 in Section 3.2). First, the specific LS pre-treatment on line 3 is the tabu list initialization. Second, the replacement strategy (line 17) is performed by the best admissible neighbor according to its availability in the tabu list. Finally, the specific post-treatment (line 18) represents the tabu list update.

Experiments have been implemented on top of three different configurations. The three GPU cards have a different number of multiprocessors (respectively 4, 16 and 30), which determines the number of active threads being executed. The number of global iterations of the tabu search is 10000 and 10 runs were performed for each instance. Time measurement is reported in seconds, and for both GPU implementation and GPU version using texture memory ($GPU_{tex}$), acceleration factors compared to a standalone CPU are designated using subindexes. Standard deviation (not represented here) is close to zero.

Experimental results for a Hamming neighborhood of distance one are depicted in Table 1 ($m$-$n$ instances). From $m = 601$ and $n = 617$, the standard GPU version starts to provide better results (from $\times 1.1$ to $\times 2.2$). Regarding the GPU version using texture memory, from $m = 301$ and $n = 317$, it starts to be faster than CPU version for both configurations (from $\times 1.4$ to $\times 1.6$). The speed-up grows with the problem size increase (up to $\times 8$ for $m = 1301$, $n = 1317$). The acceleration factor for this implementation is significant but not impressive. This can be explained by the fact that since the neighborhood is relatively small ($n$ threads), the number of threads per block is not enough to fully cover the memory access latency.

To validate this point, a neighborhood based on a Hamming distance of two on top of GPU has been implemented. Incremental evaluation is performed by a larger number of threads ($\frac{n \times (n-1)}{2}$ threads). The obtained results from

**Table 1.** Time measurements for the 1-Hamming distance neighborhood

| Instance | **Core 2X 2Ghz 8600M GT 4 multi-proc** | | **Core 4X 2.4Ghz 8800 GTX 16 multi-proc** | | **Xeon 8X 3Ghz GTX 280 30 multi-proc** | |
|---|---|---|---|---|---|---|
| | GPU | $GPU_{Tex}$ | GPU | $GPU_{Tex}$ | GPU | $GPU_{Tex}$ |
| 101-117 | $8.9_{\times 0.4}$ | $6.6_{\times 0.5}$ | $4.8_{\times 0.5}$ | $4.3_{\times 0.6}$ | $4.9_{\times 0.4}$ | $4.2_{\times 0.5}$ |
| 301-317 | $34_{\times 0.7}$ | $18_{\times \mathbf{1.4}}$ | $16_{\times \mathbf{1.1}}$ | $13_{\times \mathbf{1.5}}$ | $12_{\times \mathbf{1.4}}$ | $11_{\times \mathbf{1.6}}$ |
| 601-617 | $169_{\times \mathbf{1.1}}$ | $98_{\times \mathbf{1.9}}$ | $96_{\times \mathbf{1.4}}$ | $77_{\times \mathbf{1.7}}$ | $47_{\times \mathbf{2.2}}$ | $43_{\times \mathbf{2.4}}$ |
| 801-817 | $248_{\times \mathbf{1.5}}$ | $122_{\times \mathbf{3.1}}$ | $125_{\times \mathbf{2.1}}$ | $100_{\times \mathbf{2.7}}$ | $55_{\times \mathbf{3.6}}$ | $50_{\times \mathbf{4.0}}$ |
| 1001-1017 | $348_{\times \mathbf{1.7}}$ | $146_{\times \mathbf{4.1}}$ | $145_{\times \mathbf{3.0}}$ | $107_{\times \mathbf{4.0}}$ | $63_{\times \mathbf{5.3}}$ | $58_{\times \mathbf{5.8}}$ |
| 1301-1317 | $573_{\times \mathbf{2.1}}$ | $288_{\times \mathbf{4.1}}$ | $228_{\times \mathbf{3.4}}$ | $180_{\times \mathbf{4.3}}$ | $93_{\times \mathbf{7.4}}$ | $85_{\times \mathbf{8.0}}$ |

experiments are reported in Table 2. Due to misaligned accesses to global memories ($\epsilon$-matrix and $\epsilon$-vector) of this new neighborhood, non-coalescing memory reduces the performance of the GPU implementation on G80 series. Binding texture on global memory allows to overcome the problem. Indeed, for the first instance ($m = 101$, $n = 117$), acceleration factors of the texture version are already important (from ×4 to ×19). As long as the instance size increases, the acceleration factor grows accordingly (from ×4 to ×8.1 for the first configuration). Since a large number of multiprocessors are available on both 8800 and GTX 280, efficient speed-ups can be obtained (from ×13 to ×42.6). As a consequence, parallelization on top of GPU provides an efficient way for handling large neighborhoods.

**Table 2.** Time measurements for the 2-Hamming distance neighborhood

| Instance | **Core 2X 2Ghz 8600M GT 4 multi-proc** | | **Core 4X 2.4Ghz 8800 GTX 16 multi-proc** | | **Xeon 8X 3Ghz GTX 280 30 multi-proc** | |
|---|---|---|---|---|---|---|
| | GPU | $\mathrm{GPU_{Tex}}$ | GPU | $\mathrm{GPU_{Tex}}$ | GPU | $\mathrm{GPU_{Tex}}$ |
| 101-117 | $13_{\times 0.7}$ | $2.5_{\times \mathbf{4.0}}$ | $4.1_{\times \mathbf{1.8}}$ | $0.6_{\times \mathbf{13.0}}$ | $0.6_{\times \mathbf{12.8}}$ | $0.4_{\times \mathbf{19.0}}$ |
| 301-317 | $251_{\times 0.9}$ | $58_{\times \mathbf{4.9}}$ | $61_{\times \mathbf{2.8}}$ | $10_{\times \mathbf{16.0}}$ | $9.5_{\times \mathbf{17.8}}$ | $6.2_{\times \mathbf{27.4}}$ |
| 601-617 | $1881_{\times \mathbf{1.7}}$ | $512_{\times \mathbf{6.3}}$ | $355_{\times \mathbf{7.1}}$ | $88_{\times \mathbf{28.5}}$ | $67_{\times \mathbf{30.5}}$ | $51_{\times \mathbf{40.2}}$ |
| 801-817 | $4396_{\times \mathbf{2.0}}$ | $1245_{\times \mathbf{6.9}}$ | $815_{\times \mathbf{8.5}}$ | $210_{\times \mathbf{32.9}}$ | $152_{\times \mathbf{35.4}}$ | $128_{\times \mathbf{42.2}}$ |
| 1001-1017 | $8474_{\times \mathbf{2.1}}$ | $2502_{\times \mathbf{7.0}}$ | $1469_{\times \mathbf{9.8}}$ | $416_{\times \mathbf{34.7}}$ | $291_{\times \mathbf{38.1}}$ | $262_{\times \mathbf{42.2}}$ |
| 1301-1317 | $17910_{\times \mathbf{2.2}}$ | $4903_{\times \mathbf{8.1}}$ | $3050_{\times \mathbf{10.9}}$ | $912_{\times \mathbf{36.4}}$ | $647_{\times \mathbf{38.7}}$ | $587_{\times \mathbf{42.6}}$ |

## 7 Discussion and Conclusion

High-performance computing based on the use of GPUs is recently revealed to be a good way to accelerate computational applications. However, the exploitation of parallel models is not trivial and many issues related to GPU memory hierarchical management of this architecture have to be considered. To the best of our knowledge, GPU-based parallel LS approaches have never been deeply investigated.

In this paper, efficient mapping of the iteration-level parallel model on the GPU has been proposed according to a three-level decomposition of the GPU hierarchy. In the high-level layer, the CPU manages the whole LS process and let the GPU be used as a coprocessor dedicated to intensive calculations. Efficient mappings between neighborhood candidate solutions and GPU threads were made in the intermediate-level layer. Memory management is handled at the low-level layer. Code optimization based on texture memory is applied to the evaluation function kernel. The re-design of the parallel LS iteration-level model on GPU fits well for deterministic LS methods such as tabu search and iterated local search. Indeed, for problem instances with a large neighborhood set, the reported speed-ups provide promising results on the PPP (up to $\times 40$ with texture memory) compared to traditional CPUs. A next perspective is to apply our approach on other problems using different representations.

The approach presented in this paper might be easily extended to the variable neighborhood search heuristic, in which the same parallel exploration is applied for various neighborhoods. However, few other LS algorithms only partially explore neighborhoods and take the first improving local neighbor that is detected. Applied to LS algorithm such as simulated annealing, this model needs to be re-thought.

## References

1. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using cuda. J. Parallel Distributed Computing **68**(10) (2008) 1370–1380
2. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO. (2007) 1566–1573
3. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems **22**(2) (2007) 69–78
4. Banzhaf, W., Harding, S.: Accelerating evolutionary computation with graphics processing units. In: GECCO (Companion). (2009) 3237–3286
5. Pointcheval, D.: A new identification scheme based on the perceptrons problem. In: EUROCRYPT. (1995) 319–328
6. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., mei W. Hwu, W.: Program optimization carving for gpu computing. J. Parallel Distribibuted Computing **68**(10) (2008) 1389–1401
7. Talbi, E.G.: Metaheuristics: From design to implementation. Wiley (2009)
8. NVIDIA: CUDA Programming Guide Version 2.1. (2009)
9. Luong, T.V., Melab, N., Talbi, E.G.: Parallel Local Search on GPU. Research Report RR-6915, INRIA (2009)