A GPU Implementation of Parallel Constraint-based Local Search

Alejandro Arbelaez[1] and Philippe Codognet[2]

[1] JFLI / University of Tokyo
[2] JFLI - CNRS / UPMC / University of Tokyo
University of Tokyo, Dept. of Computer Science,
7-3-1, Hongo, Bunkyo-ku, 113-0033 Tokyo, Japan
{arbelaez,codognet}@is.s.u-tokyo.ac.jp

Abstract. In this paper we study the performance of constraint-based local search solvers on a GPU. The massively parallel architecture of the GPU makes it possible to explore parallelism at two different levels inside the local search algorithm. First, by executing multiple copies of the algorithm in a multi-walk manner and, second, by evaluating large neighborhoods in parallel in a single-walk manner. Experiments on three well-known problem benchmarks indicate that the current GPU implementation is up to 17 times faster than a well-tuned sequential algorithm implemented on a desktop computer.

1 Introduction

In the last decade, the interest for the family of Local Search methods and Metaheuristics for solving large combinatorial problems has been growing and has attracted much attention from both the Operations Research and the Artificial Intelligence communities for solving real-life problems [1]. Theses methods have been used in Combinatorial Optimization for finding optimal or near-optimal solutions for several decades and they are now widely used to solve real-life problems when the search space is too large to be explored by complete search algorithm, such as Mixed Integer Programming or Constraint Solving. There also exist some efficient general-purpose systems for Local Search, such as for instance the commercial system Comet [2].

In general there exists two approaches to devise a parallel local search solver: multi-walk and single-walk [3]. Multi-walk methods consist in developing concurrent explorations of the search space, either independently or cooperatively with some communication between the processes. Sophisticated cooperative strategies for multi-walk methods can be devised by using solution pools [4], but require shared-memory or emulation of central memory in distributed clusters, thus impacting on performance. On the other hand, Single-walk methods consist in using parallelism inside a single search process, *e.g.,* for parallelizing the exploration of the neighborhood.

Graphic Processing Units (GPUs) are nowadays available in nearly all personal computers and they offer a potential reduction in the computational time

of local search solvers. The *thread hierarchy* in a GPU consists of threads, blocks, and grids. A block is a batch of threads (all executing the same code) and blocks are grouped in a grid (blocks are independent). This hierarchy matches the architecture of the local search process by allowing the combination of multi-walk and single-walk. Taking this into account, independent local search algorithms can be allocated in different blocks, each block exploiting parallelism by evaluating neighbors in parallel.

The rest of the paper is structured as follows. Section 2 presents a generic description of local search and the *Adaptive Search* method; a generic, domain-independent constraint-based local search algorithm. Section 3 presents a description of parallel local search and the most relevant CPU and GPU implementations in the area. Section 4 details the architecture of modern GPUs. Section 5 highlights the main considerations to take into account when designing a local search algorithm targeting a GPU. Section 6 details the problems benchmarks used in this paper. Section 7 presents experiments of the CPU and GPU implementations. A conclusion and discussion of future work end the paper.

2 Local Search

Local Search algorithms start from an initial assignment for the variables (usually random) and try to improve this configuration, little by little, by small changes in the values of the problem variables. Hence the term "local search" as, at each time step, only new configurations that are "neighbors" of the current configuration are explored. The definition of what constitutes a neighborhood will of course be problem-dependent, but basically it consists in changing the value of a few variables only (usually one or two). The advantage of Local Search methods is that they will usually quickly converge towards a solution (if the optimality criterion and the notion of neighborhood are defined correctly) and not exhaustively explore the entire search space. This is however at the loss of completeness of the search. These methods naturally lead to concurrent execution, by considering the development of several configurations at the same time. This can be done sequentially by maintaining a pool of candidate configurations (as in genetic algorithms) or in parallel if the adequate hardware is available.

The domain of Constraint-based local search, that is, applying local search techniques to solve Constraint Satisfaction Problems has been attracting some interest for about a decade [5,6,2]. Constraint Satisfaction can obviously be seen as a branch of Combinatorial Optimization in which the objective function to minimize is the number of violated constraints: a solution is therefore obtained when the function has value zero. Constraint-based local search can tackle large size problem instances, far beyond the reach of classical propagation-based constraint solvers.

2.1 The Adaptive Search Method

Algorithm 1 presents the general scheme of the *Adaptive Search* (AS) model, a generic constraint-based local search method proposed nearly a decade ago [5,7,8].

This meta-heuristic takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a single global cost function to optimize, such as for instance the number of violated constraints. The algorithm also uses a short-term adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. Moreover it intrinsically copes with over-constrained problems.

The input of the method is a Constraint Satisfaction Problem (CSP), which is defined as a triple (X;D;C), where X is a set of variables, D is a set of domains, *i.e.,* finite sets of possible values (one domain for each variable), and C a set of constraints restricting the values that the variables can simultaneously take. For each constraint, an *error function* needs to be defined; it gives, for each tuple of variable values, an indication of how much the constraint is violated. This idea has also been proposed independently by [6], where it is called "penalty functions", and then reused by the Comet system [2], where it is called "violations". For example, the error function associated with an arithmetic constraint $|X - Y| < c$, for a given constant $c > 0$, can be $max(0, |X - Y| - c)$.

AS relies on iterative repair, based on variable and constraint error information, seeking to reduce the error on the variable with the highest error value. The basic idea is to compute the error function for each constraint, then combine for each variable the errors of all constraints in which it appears, thereby projecting constraint errors onto the relevant variables. This combination of errors is problem-dependent, see [5] for details and examples, but it is usually a simple sum or a sum of absolute values, although it might also be a weighted sum if constraints are given different priorities. Finally, the variable with the highest error is designated as the "culprit" and its value is modified. In this second step, the well known min-conflict heuristic [9] is used to select the value in the variable domain which is the most promising, that is, the value for which the total error in the next configuration is minimal.

In order to prevent being trapped in local minima, the AS method also includes a short-term memory mechanism to store configurations to avoid (variables can be marked Tabu and "frozen" for a number of iterations). It also integrates reset transitions to escape stagnation around local minima. A reset consists in assigning fresh random values to some variables (also randomly chosen). A reset is guided by the number of variables being marked Tabu. It is also possible to restart from scratch when the number of iterations becomes too large (this can be viewed as a reset of all variables but it is guided by the number of iterations). The core ideas of adaptive search can be summarized as follow:

- to consider for each constraint a heuristic function that is able to compute an approximated degree of satisfaction of the goals (the current *error* on the constraint);
- to aggregate constraints on each variable and project the error on variables thus trying to repair the *worst* variable with the most promising value;
- to keep a short-term memory of bad configurations to avoid looping (*i.e.* some sort of *tabu list*) together with a reset mechanism.

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:
- variables X_i with their domains
- constraints C_j w/error functions
- function to project errors on vars
- cost function to minimize

some tuning parameters:
- TT: # iterations a variable is frozen
- RL: # frozen variables triggering a reset
- RP: % of variables to reset
- MI: max. # iterations before restart
- MR: maximal # of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

```
1: Restart ← 0
2: repeat
3:     Restart ← Restart + 1
4:     Iteration ← 0
5:     Compute a random assignment A of variables in V
6:     Opt_Sol ← A
7:     Opt_Cost ← cost(A)
8:     repeat
9:         Iteration ← Iteration + 1
10:        Compute errors of all constraints in C and combine errors on each variable
11:                ▷ (by considering only the constraints in which a variable appears)
12:        Select the variable X (not marked Tabu) with highest error
13:        Evaluate costs of possible moves from X
14:        if no improvement move exists then
15:            mark X as Tabu until iteration number: Iteration + TT
16:            if the number of variables marked Tabu ≥ RL then
17:                randomly reset RP % variables in V (and unmark those Tabu)
18:            end if
19:        else
20:            Select best move and change X, yielding the next configuration A'
21:            if cost(A') < Opt_Cost then
22:                Opt_Sol ← A ← A'
23:                Opt_Cost ← cost(A')
24:            end if
25:        end if
26:    until Opt_Cost = 0 (a solution is found) or Iteration ≥ MI
27: until Opt_Cost = 0 (a solution is found) or Restart ≥ MR
28: output(Opt_Sol, Opt_Cost)
```

A few improvements to the basic algorithm are important and worth mentioning here, let us just briefly detail the two main ones. Firstly, when following a *plateau*, a simple stochastic scheme is used for deciding either to continue on the plateau or to escape, by adding a probability p for doing this. With good tuning (*e.g.,* probability of 90% to 95% of following a plateau) this boosts the performance of the algorithm by an order of magnitude on some problems such as Magic-Square. Secondly, when too many variables become Tabu, there is a risk of "freezing" the configuration and of getting trapped around a local minimum. Diversification is then performed by a (partial) reset, *i.e.,* by assigning fresh values to a given percentage of the problem variables (parameter RP of the

algorithm). A reset is triggered by the total number of variables being marked Tabu at a given iteration (parameter RL of the algorithm).

AS is a simple algorithm but it turns out to be quite efficient in practice. A C-based implementation of AS for permutation problems is available at http://cri-dist.univ-paris1.fr/diaz/adaptive/. We used this version as a baseline of the GPU implementation. In [10] AS is briefly compared with Comet [2] on some CSP benchmarks included in the distribution of Comet such as the N-QUEENS problem and the MAGIC-SQUARE problem. Adaptive Search is one or two order of magnitude faster than Comet. Of course, it should be noticed that Comet is a complete and very versatile system while AS is just a C-based library. AS is also compared on the COSTAS ARRAY problem in [11] with Dialectic Search [12] and it is nearly one order of magnitude faster.

3 Parallel Local Search

As pointed out above, there are two main parallel techniques to build parallel local search solvers. On one hand, the parallel speedup of the single-walk approach is limited by the Ambahl's law. We recall that the Amdahl's low indicates that the parallel speedup of a given algorithm is bounded by the sequential portions of the code. For instance, an algorithm with 5% of sequential code has a maximum speedup of factor 20. On the other hand, the speedup of the multi-walk method highly depends on the sequential behavior of the algorithm. It is well-known that if the runtime distribution of a local search algorithm follows a pure exponential law (i.e., non-shifted), its parallel multi-walk version will have a linear speedup for an unbounded number of cores [3]. However, in practice, only very few examples have a runtime distribution following a such an ideal case and the speedup is far from linear for a large number of cores [13,14].

3.1 Constraint-based Parallel Local Search

Most of the work in parallel constraint-based parallel local search for traditional architectures (*i.e.,* PC clusters) has been devoted to using the multi-walk method, mainly because this method provides two general advantages. First, it requires no extra work to implement, and second it has been theoretically and practically proven to be powerful in a wide range of domains; moreover, this technique is not affected by the Amdahl's law.

A multi-walk adaptation of AS has been proposed recently and has been implemented on massively parallel computers and grid systems [10], Cell Broadband Engine [15], and devices with Partitioned Global Address Space in [16]. Although for a few problems such as the COSTAS ARRAY parallel speedup can be linear (experiments on the BlueGene supercomputer show linear speedup up to 8000 cores, see [17]), for many problems speedup are good but suboptimal. There is therefore a need for more complex parallelization schemes, for example that would mix single-walks and multi-walks, *i.e.,* parallelizing both neighborhood search and multi-starts.

3.2 Local Search on GPUs

In [18] the authors use the GPU to accelerate the resolution of a variety of problems such as Quadratic Assignment Problem and Traveling Salesman Problem. The authors propose the use of texture memory to store the definition of the problem and perform the execution of the local search algorithm on both the CPU and the GPU. The CPU is in charge of the control of the algorithm, and the GPU evaluates a rather expensive objective function for a large number of neighbors. This approach allows the exploration of the multi-walk parallel algorithm by executing different copies of the hybrid algorithm on multiple GPUs, *i.e.,* one local search process per GPU. The algorithm reports speedups (w.r.t. to the sequential CPU implementation) of up to 50 for traditional benchmark instances and up to 240 for benchmark instances that required vector of real values.

In the same direction as [18], other authors also use the GPU to accelerate the performance of local search algorithms by means of exploring very large neighborhoods (see [19] for a recent survey). For instance, [20] indicate that the GPU+CPU algorithm reports speedups up to a factor of 27 w.r.t. the sequential implementation for the unrelated parallel machine scheduling problem. [21] uses the GPU to efficiently explore neighborhoods with millions of elements for the vehicle routing problem.

In [22] the authors proposes a framework for heterogeneous systems with multiple CPUs and GPUs interconnected through a network. The framework executes several copies of the local search algorithm and allows cooperation by sharing the best solution found so far for each algorithm. This information is then exploited at each restart to properly craft a new assignment for the variables to start with. The authors present extensive experimental results targeting the TSP problem which indicate that their approach scale up to thousands of cores (including 256 CPUs and 384 GPUs).

In a different context, [23] uses the GPU to solve SAT instances using a tree-based search algorithm, the algorithm combines backtracking search and clauses learning. Also in the context of SAT solving, in [24] the authors implement the survey propagation algorithm on a GPU and showed that the GPU version is up to 9 times faster than the sequential algorithm implemented in the CPU.

4 The GPU Architecture

Figure 1 describes the architecture of the GPU. This architecture offers an interesting opportunity to improve the performance of parallel local search solvers. Unlike a CPU which usually consists of few processors units (usually dual or quad cores), a GPU consists of a set of Stream Multiprocessors (SMs), each one features multiple processing units with the ability of executing thousands of operations concurrently. Inside a SM, threads are grouped into warps. A warp executes 32 threads in a SIMD (Single Instruction, Multiple Data) manner, that is all threads within a warp execute the same operation on multiple data points.

Furthermore, warps are grouped into a block and blocks are grouped into a grid. NVIDIA offers the Cuda Occupacy Calculator tool which allows to determine the maximum number of active blocks per grid, this number varies for a number of reasons including: memory limitations, maximum number of threads per block, and hardware limitations.

Generally speaking, a GPU uses two types of memory: global and shared. Shared memory is usually limited to few KiloBytes (*e.g.,* 16KB or 48KB) per SM which is equally divided into all blocks allocated for a given SM; an important feature of shared memory is that it has low latency requiring about 2 clock cycles to issue a memory instruction. On the other hand, global memory allows to store a large amount of data (*e.g.,* 6GB), however, it has high latency and requires between 400 and 600 clock cycles to issue a memory instruction. Moreover, GPUs are also equipped with texture and constant memory sections for read-only operations; accessing these two memories is usually faster than the global memory as they include on-chip cache.

Branch divergence and coalesced memory are among the most important aspects to take into consideration when designing algorithms in a GPU. First, branch divergence refers to the fact that threads within the same warp can take different paths (*e.g., if-else* instructions), in this case different execution paths are serialized. In the worst case-scenario a warp executes one thread a time until completing the instruction. Second, in order to obtain peak performance, accesses to global memory should be coalesced, that is, individual threads must access independent and consecutive elements from the global memory; non-coalesced accesses to global memory might result in an important degradation in the performance of the algorithm.

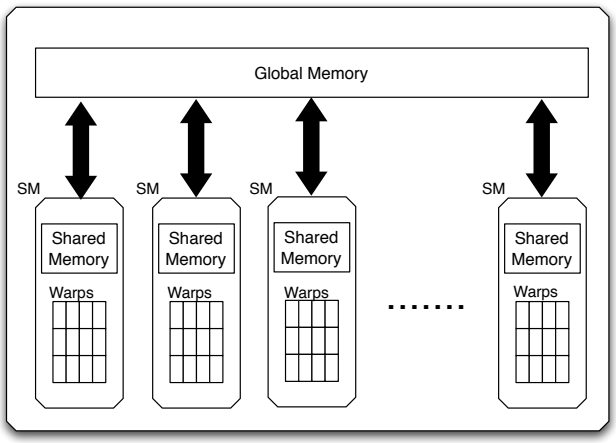


Fig. 1. GPU Architecture

5 Constraint-based Local Search on a GPU

The thread hierarchy of the GPU matches the architecture of the local search algorithm depicted in Section 2 and allows to exploit parallelism at two different levels. Firstly, multi-walk parallelism can be achieved by executing multiple copies of Algorithm 1 in parallel using different blocks. This form of parallelization is probably the most widely used one due to its simplicity and it often provides a great speedup up to few hundreds of cores. Secondly, the parallel nature of the GPU also allows to speedup the search process by including parallelism inside the blocks in a single-walk manner; this form of parallelism is also known as parallel neighborhood evaluation. In the context of AS, it would consist in selecting the two most suitable variables for swapping.

Broadly speaking, AS employs two strategies for selecting the variables to swap. The first one (*SelVars-A*), or exhaustive search, evaluates all possible swaps at a given state of the search, and selects uniformly at random, the best one. This method explores a large neighborhood, which usually grows quadratically with the size of the problem. The second one (*SelVars-B*), employs a heuristic strategy to select the best move, it starts by selecting the variable X_i with the highest error and then selects another variable X_j that when swapping X_i and X_j minimizes the global error; this method requires a linear complexity $O(n)$, where n is the number of variables of the problem. In both situations, the goal is to select the best move in a considerably large space.

At a first glance, one could consider applying the parallel reduction technique [25], which reduces the complexity to $O(log\ n)$ by applying a tree decomposition of the problem; and returns the best candidate breaking ties in a rather deterministic way. However, a key point to allow diversification in the local search procedure consists in breaking ties with an uniform distribution, then all candidates are equally likely to be selected. Taking this into account, we decided to use the traditional divide-and-conquer approach, where the problem space is divided into several independent subspaces and select, in parallel, the best candidate for each sub-space at a cost of reconciling partial information in order to decide the best action. This simple approach reduces the complexity of an iteration to $O(n/m + m)$, where m is the number of threads per block.

As pointed out in the previous section, an important aspect to take into consideration when designing algorithms in a GPU is the administration of the global and shared memory. Ideally, it would be desirable to store all relevant information in shared memory. However, a CSP might contain thousands of variables and constraints, and it would require too much memory. In a nutshell, the main aspects to consider in the *Adaptive Search* framework are the following:

- Current assignment: requires a linear space complexity and represents the current value for the variables.
- Tabu list: a common way to implement a Tabu list consists in maintaining a list with the age of the variables (indicating the last time it was used). Thus, a variable is Tabu if it has been used within the last $tabuTenure$ iterations.

– Incremental update: a critical aspect when implementing efficient local search algorithms is to incrementally update the objective function. As will be described in Section 6 for a few examples, the *Adaptive Search* framework requires a set of extra variables to maintain an efficient incremental evaluation of the error function.
– List of best candidate variables: when selecting the swapping variables, it is necessary to maintain a list with the variables that maximizes (or minimizes) the objective function, then a variable from the list is chosen with an uniform distribution.

The decision of using the global or shared memory is entirely up to the problem. In a few cases, it would be possible to only use the shared memory; however, as will be detailed in Section 6, to tackle large and complex instances it would necessary to extensively use the global memory.

6 Problem Benchmarks

In this section, we detail the benchmarks used for comparison and provide a brief description of how they are implemented in the GPU. One key issue is to reduce accesses to global memory in order to avoid degrading performance of the GPU implementation.

Magic Square

This is problem 19 of CSPLib[3]. It consists in completing a $N \times N$ matrix with numbers $1, 2, \ldots, N^2$, where each column, row, and the two main diagonals sum the same objective number b. As described in [5], the *Adaptive Search* model of this problem is an instance of magic square in which $b{=}N(N^2+1)/2$. The error function for each constraint is defined as $f(X){=}X_1{+}X_1{+}\ldots{+}X_k$ - b, and the global error is the sum of the absolute value of the error for all constraints.

For an efficient incremental evaluation of the global error, the algorithm maintains a list of $2n{+}2$ elements with the error of the columns, rows, and the two diagonals. This way, obtaining the error of swapping two variables is then done in constant time. AS uses *SelVars-B* to identify the most suitable variables to swap at each iteration of the algorithm; therefore, the algorithm explore a neighborhood of N^2 elements, where N is the input of the problem.

We decided to keep the list variables with the errors for columns, rows, and the two diagonals in shared memory, and the remaining variables in global memory. The GPU implementation of the magic square problem uses the same parameter configuration as the CPU version, except that in order to reduce accesses to global memory, the GPU implementation does not clean the Tabu list after performing a reset.

[3] www.csplib.org

Number Partition

This is problem 49 of CSPLib. It consists in dividing a set of numbers from 1 to N into two subsets, where both subsets have the same cardinality, the same sum of numbers, and the same sum of squares of numbers. The model of this problem involves two set of $N/2$ variables each one, and the global error is obtained as the sum the absolute value of the errors of the two following constraints:

$$\sum_{i=1}^{N/2} X_i = \sum_{i=N/2+1}^{N/2} X_i = N(N+1)/4$$
$$\sum_{i=1}^{N/2} X_i^2 = \sum_{i=N/2+1}^{N/2} X_i^2 = N(N+1)(2N+1)/12$$

At each iteration of the local search process the algorithm uses *SelVars-A* (or exhaustive search), and explores all possible swaps by exchanging values from the two subsets. Thus, the algorithm needs to explore a neighborhood of $(N/2)^2$ elements. For this problem, the shared memory is used for the list of variables representing the current configuration, and the global memory stores the remaining variables. The GPU version of the algorithm uses the same parameters as the CPU version, except that the CPU uses a Tabu list of size one, and the GPU algorithm omits the Tabu list in order to reduce accesses to global memory.

Costas Array

Costas array is a hard combinatorial problem abstracted from radar and sonar applications [26]. It consists in placing N marks in a $N \times N$ grid, such that there is exactly one mark per row, column, and the $N(N-1)/2$ vectors joining the marks are all different. This problem is modeled using the difference triangle, in which N variables form a permutation of numbers from 1 to N, and the $j-th$ row of the triangle contains the differences between the variables X_{i+j} - X_i, where $j \in [1, N-1]$ and $i \in [1, N-j]$. In order to obtain the cost of a given assignment for the variables the algorithm computes the global error by checking the first $(N-1)/2$ rows of the triangle. In this case, AS uses *SelVar-B* to identify the best move at each iteration of the local search process. Since this problem requires a rather small number of variables and constraints, we decided to store all variables in shared memory.

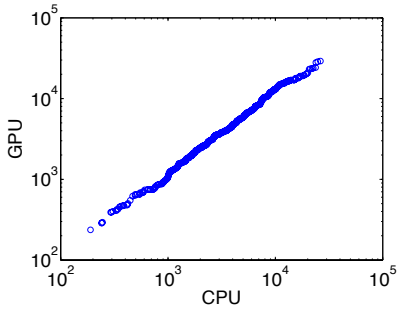
7 Experiments

This section compares the performance of the CPU and GPU implementations of AS. All the experiments were performed using a 6-core processor (Xeon W3670 at 3.20 Ghz) and 12MB of RAM, and a GPU NVIDIA TESLA C2075 which features 14 SMs, 6GB of global memory and up to 48KB of shared memory per SM. Although this GPU allows up to 1536 threads per block, the following experimentation is limited to up to 512 threads per block due to a limitation in the number of registers of the SMs. We implemented the GPU algorithm using CUDA version 5. For all the experiments we used the default reset function

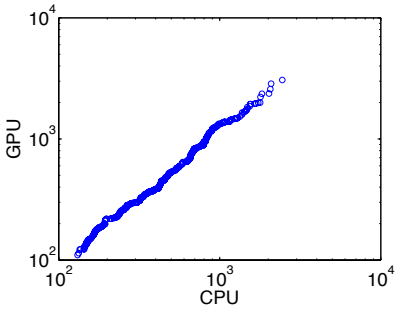
described in Section 2, and in the following results we report the median value of 20 executions unless otherwise stated.

We start our analysis with Figure 2, which depicts the quantile-quantile plot (Q-Q plot) of the number of iterations required to solve three small size instances of Magic Square (20), Partition (2600), and Costas Array (15) for the CPU and GPU implementations of AS. The Q-Q plot displays the quantiles of the data of the two samples of 500 runs. A given *quantile* is a point in a sample for which a certain fraction of the sample lies, *e.g.,* the 0.3 quantile is the point for which 30% falls below and 70% above. For example the well-known *median value* of a sample is the 0.5 quantile (also noted 50% quantile). A *quantile-quantile plot* will depict, for the same quantile, the value in one sample as abscissa and the value of the other one as ordinate. Thus, checking if both samples match is straightforward: if points are on the diagonal ($y = x$), then both samples perfectly fit. The dimension of both axes are the number of iterations.

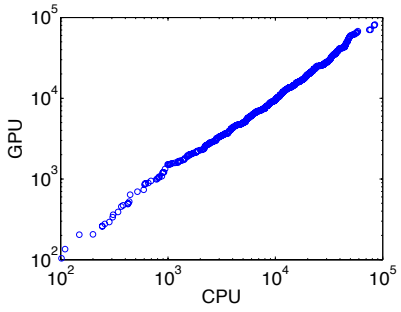
Although the GPU implementation reuses large portions of the original algorithm, the figure shows that the two versions behave very similarly but are not perfectly identical. This is due the slightly differences indicated the previous section to provide an efficient GPU algorithm.



(a) Magic Square 20



(b) Partition 2600



(c) Costas 15

Fig. 2. Q-Q plot of the CPU and GPU implementations for three different CSPs based on 500 runs per algorithm.

Tables 1 & 2 report the runtime in seconds to solve Magic Square problem (N=200, 300 and 400) on the GPU using different configurations for the grid, *i.e.,* varying the ratio threads/block. We set the minimum number of threads per block to 32 as this is minimum allowed by CUDA and the number of blocks varies from 1, 7, 14, 28, and 56. Bold number indicate the best performance obtained in this table. As expected, we observe a reduction in the runtime by only using the multi-walk parallel scheme (columns), however, the speedup changes when increasing the number of threads per block. For instance, when using 32 threads per block, we observe a maximum speedup of 9.46 (1 block vs. 56 blocks), but when using 512 threads per block, we observe a maximum speedup of 3.16. There are several aspects that influence the speedup of the multi-walk algorithm in the GPU, such as divergent branch, non-coalesced memory, bandwidth occupancy, warp synchronization, etc. We recall that such speedup differences should not be observed in homogeneous cluster with nodes that have similar computation capabilities. For these reasons, increasing the number of threads per block might degrade the performance of the multi-walk parallel algorithm.

Table 1. GPU implementation of Magic Square 200 (time in secs)

Blocks \ Threads	32	64	128	256	512
1	492	280	96	60	79
7	116	56	48	26	17
14	59	35	28	20	**16**
28	64	64	23	27	18
56	52	24	37	39	25

The best performance with the GPU for Magic Square 200 is 16 secs, this value is obtained when using 14 blocks and 512 threads per block. We attribute this optimal value to the fact that the reference GPU uses 14 SM. On the other hand, it also important to notice that this optimal performance obtained with the GPU is 4.5 times faster than the sequential CPU version (see Table 3) and 1.5 times faster than the best runtime obtained with the CPU using 6 cores. Furthermore, the best result obtained for Magic Square 300 (54 secs) is 17 times faster than the sequential algorithm, and 1.8 times faster than the best CPU multi-core implementation (with 6 cores). And for Magic Square 400 the best result of the GPU (160 secs) is 16 times faster than the sequential CPU version[4], and 3 times faster than the best CPU multi-core performance (with 6 cores). We believe that increasing the size of the problem might also increase the difference in the performance between the GPU and the CPU. At this point it is worth noticing that parallel versions of local search algorithms can achieve super-linear speedups w.r.t. the sequential algorithm (*e.g.,* CPU implementation of Magic

[4] The runtime of the CPU sequential version of Magic Square (N=400) was computed as the median value of 11 executions

Square 300). This is the case for instance when the sequential runtime distribution follows a lognormal distribution, which is heavy-tailed. This is because the sequential algorithm exhibits the heavy tail phenomena, this phenomena is less relevant when executing the algorithm in parallel [14].

Table 2. GPU implementation of Magic Square 300 and 400 (time in secs)

Size	Blocks \ Threads	128	256	512
300	14	107	75	**54**
	28	76	80	79
400	14	298	**160**	273
	28	191	279	209

Table 3. CPU implementation of Magic Square (time in secs)

Cores \ Size	200	300	400
1	72	970	2572
3	35	115	710
6	**24**	**98**	**483**

Interestingly, the performance using a single GPU outperforms a multiple-walk implementation of AS for the Magic Square problem (size 400) on a supercomputer using 64 cores, *i.e.,* 198 secs (Hitachi HA8000 Supercomputer with 64 cores) vs. 160 secs (GPU with 14 blocks and 256 threads per see [10].

Let us now switch our attention to the Partition problem (size 9600). Table 4, this table depicts the experimental results of the GPU to solve . In this case, we observe that the best performance is 6 secs, obtained when using 14 blocks and 512 cores. On the other hand, the GPU reports a speedup of 17 w.r.t. the sequential implementation, and a speedup of 5 against the best parallel CPU implementation (with 6 cores).

Finally, Tables 7 and 6 show the results of two instances of the Costas array problem (*i.e.,* N=18 and N=19). Unlike Magic Square and partition, which involve the exploration of large neighborhoods (quadratic w.r.t. the size of the problem), the Costas array problem requires to explore a small neighborhood of N elements, where N is usually a small number. Therefore, for this problem we increase the number of blocks up to 112 (being 144 the maximum number of blocks allowed for this GPU, *i.e.,* 12 blocks per SM). In this case, the best performance observed for the GPU is about 2 times faster than the sequential CPU implementation for Costas 18, and about 3 times faster for Costas 19; however, due to the great speedup capabilities of the AS to solve the Costas

Table 4. GPU implementation of Partition 9600 (time in secs)

Blocks \ Threads	32	64	128	256	512
1	521	279	104	123	84
7	68	45	32	9	9
14	37	28	16	9	**6**
28	37	30	14	10	9
56	43	30	12	8	9

Table 5. CPU implementation of Partition (time in secs)

Cores \ Size	9600
1	102
3	49
6	30

array problem the best parallel CPU implementation (with 6 cores) is faster than the GPU one.

Even though the parallel multi-core implementation for Costas is better than the GPU one, these results are encouraging due to the fact that for this problem the GPU is being underused. Notice that for this problem the algorithm does not exploit parallelization in a single-walk manner. Indeed, part of our future work will be to support the multi-walk method at a warp level, *i.e.,* executing multiple local search processes inside a single block.

Table 6. GPU Implementation of Costas array (time in secs)

Size \ Blocks	1	7	14	28	56	112
18	204	34	22	10	8	**5**
19	1453	189	128	72	33	**21**

8 Conclusions and Future Work

In this paper, we have presented a GPU implementation of the *Adaptive Search* framework. This new version of *Adaptive Search* exploits the architecture of the GPU to device a parallel local search algorithm. Thus, the algorithm exploits parallelism by executing multiple local search processes and at the same time allows the exploration of the large neighborhoods in parallel. Unlike other local search implementations which are problem dependent and combine the execution of the CPU and the GPU, in this work the entire local search process is executed

Table 7. CPU implementation of Costas Array (time in secs)

Size \ Cores	1	3	6
18	10	4	**0.3**
19	66	24	**19**

on the GPU; therefore, the performance is not bounded to a robust CPU. Extensive experiments using three well-known problem benchmarks indicate that the GPU provides speedups up to 17 w.r.t. a well tuned sequential version of the Magic Square and Partition problems, and up to 3 for the sequential CPU version of the Costas Array problem. Furthermore, we believe that increasing the input size for the problems might also increase the difference between the CPU and the GPU.

The current GPU version of AS allows the execution of only one local search process per block, and limits the number of threads for the parallel evaluation of the neighborhood to 512 (max. number of threads per block); however, different problems might require more computational power. For instance, more threads to evaluate the neighborhood (*e.g.,* Partition and Magic Square), and other problems can exploit better the parallelization by executing multiple local search processes per block (*e.g.,* Costas array). We also plan to study very large neighborhoods using the GPU, we recall that the current version is an adapted version of CPU implementation, however, the parallel nature of the GPU allows more flexibility. Furthermore, we plan to extend the current implementation to support execution on multiple GPUs and clusters of GPUs. Lastly, we believe that there is significant room for improving the performance of the parallel algorithm by means of including cooperation between the solvers. Local search solvers can exploit parallelization by exchanging information such as the best assignment for the variables found so far, and using this information to push the trajectory of the algorithm to promising areas of the search space [27].

9 Acknowledgements

The first author was supported by the Japan Society for the Promotion of Science (JSPS) under the JSPS Postdoctoral Program and the *kakenhi* Grant-in-aid for Scientific Research.

References

1. Gonzalez, T., ed.: Handbook of Approximation Algorithms and Metaheuristics. Chapman and Hall / CRC (2007)
2. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
3. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics **1**(1) (1995) 43–65

4. Crainic, T.G., Gendreau, M., Hansen, P., Mladenovic, N.: Cooperative parallel variable neighborhood search for the -median. Journal of Heuristics **10**(3) (2004) 293–314
5. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: proceedings of SAGA'01, Springer Verlag (2001) 73–90
6. Galinier, P., Hao, J.K.: A general approach for constraint solving by local search. In: 2nd workshop CP-AI-OR'00, Paderborn, Germany (2000)
7. Codognet, P., Diaz, D.: An efficient library for solving CSP with local search. In Ibaraki, T., ed.: MIC'03, 5th International Conference on Metaheuristics. (2003)
8. Newton, M.A.H., Pham, D.N., Sattar, A., Maher, M.J.: Kangaroo: An efficient constraint-based local search system using lazy propagation. In Lee, J.H.M., ed.: CP. Volume 6876 of Lecture Notes in Computer Science., Springer (2011) 645–659
9. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence **58**(1-3) (1992) 161–205
10. Caniou, Y., Codognet, P., Diaz, D., Abreu, S.: Experiments in parallel constraint-based local search. In Merz, P., Hao, J.K., eds.: EvoCOP'11. Volume 6622 of LNCS., Torino, Italy, Springer (April 2011) 96–107
11. Diaz, D., Richoux, F., Codognet, P., Caniou, Y., Abreu, S.: Constraint-based local search for the costas array problem. In: LION 6, Learning and Intelligent OptimizatioN Conference, Paris, France, Springer LNCS (2012)
12. Kadioglu, S., Sellmann, M.: Dialectic search. In: CP'09, 15th Int. Conf. on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, Springer Verlag (2009)
13. Arbelaez, A., Truchet, C., Codognet, P.: Using sequential runtime distributions for the parallel speedup prediction of sat local search. Journal Theory and Practice of Logic Programming (TPLP) special issue, proceedings of ICLP13, to Appear.
14. Truchet, C., Richoux, F., Codognet, P.: Prediction of Parallel Speed-ups for Las Vegas Algorithms. In Dongarra, J., Robert, Y., eds.: Proceedings of ICPP-2013, 42nd International Conference on Parallel Processing, IEEE Press (October 2013)
15. Diaz, D., Abreu, S., Codognet, P.: Parallel constraint-based local search on the cell/be multicore architecture. In: proceedings of IDC2010, Intelligent Distributed Computing IV, Springer Verlag (2010)
16. Machado, R., Abreu, S., Diaz, D.: Parallel local search: Experiments with a pgas-based programming model. In: International Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS), Budapest, Hungary (2012)
17. Diaz, D., Richoux, F., Caniou, Y., Codognet, P., Abreu, S.: Parallel local search for the costas array problem. In: IEEE Workshop on new trends in Parallel Computing and Optimization (PC012), in conjunction with IPDPS 2012, Shanghai, China, IEEE Press (May 2012)
18. Luong, T.V., Melab, N., Talbi, E.G.: Gpu computing for parallel local search metaheuristic algorithms. IEEE Trans. Computers **62**(1) (2013) 173–185
19. Talbi, E.G., Hasle, G.: Metaheuristics on gpus. J. Parallel Distrib. Comput. **73**(1) (2013) 1–3
20. Coelho, I.M., Haddad, M.N., Ochi, L.S., Ochi, L.S., Farias, R.: A hybrid cpu-gpu local search heuristic for the unrelated parallel machine scheduling problem. In: 3rd workshop on Applications for Multi-Core Architectures (WAMCA). (2012) 19–23
21. Schulz, C.: Efficient local search on the gpu - investigations on the vehicle routing problem. J. Parallel Distrib. Comput. **73**(1) (2013) 14–31

22. Burtscher, M., Rabeti, H.: A scalable heterogeneous parallelization framework for iterative local searches. In: IPDPS'13. (2013)
23. Palu, A.D., Dovier, A., Formisano, A., Pontelli, E.: Exploiting unexploited computing resources for computational logics. In: 9th Italian Convention on Computational Logic. (2012)
24. Manolios, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In: SAT. (2006) 311–324
25. Blelloch, G.E.: Scans as primitive parallel operations. IEEE Trans. Computers **38**(11) (1989) 1526–1538
26. Drakakis, K.: A review of costas arrays. Journal of Applied Mathematics **2006** (2006) 1–32
27. Arbelaez, A., Codognet, P.: Massivelly Parallel Local Search for SAT. In: ICTAI'12, Athens, Greece, IEEE Computer Society (November 2012) 57–64