A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems

F. Campeotto and **A. Dovier** and **F. Fioretto** and **E. Pontelli** [1]

Abstract. Constraint programming has gained prominence as an effective and declarative paradigm for modeling and solving complex combinatorial problems. In particular, techniques based on local search have proved practical to solve real-world problems, providing a good compromise between optimality and efficiency. In spite of the natural presence of concurrency, there has been relatively limited effort to use novel massively parallel architectures—such as those found in modern Graphical Processing Units (GPUs)—to speedup constraint programming algorithms, in general, and local search methods, in particular. This paper describes a novel framework which exploits parallelism from a popular local search method (the *Large Neighborhood Search* method), using GPUs.

1 Introduction

Constraint programming (*CP*) is a declarative paradigm designed to provide high-level modeling and resolution of combinatorial search problems. It is attracting widespread commercial interest and it is now becoming the method choice for modeling many types of optimization problems (e.g., [11, 19, 2]), possibly combined with other techniques. A problem is modeled using variables, each of them coming with its own domain (typically, a finite set of values), and a set of *constraints* (i.e., relations) among variables. The model is given as input to a constraint tool (e.g., [17, 10]), which explores the search space of possible solutions, alternating non-deterministic variable assignments and deterministic constraint propagation. The goal is typically to find one (all) solution(s), the best one, or determine that the problem is unsatisfiable. The programmer might help the search by tuning search parameters or providing problem-specific knowledge.

Although this *declarative* approach allows one to model a broad class of optimization problems with relatively ease, real-world combinatorial optimization problems are often characterized by huge search spaces (e.g., [6]) and heterogeneous sets of constraints. If this is the case, *incomplete* search strategies (e.g., local search strategies) are usually preferred to exact approaches that would require prohibitive time to determine an optimal solution.

Recent technological trends have made highly parallel hardware platforms and corresponding programming models available to the broad users community—transforming high performance computing from a specialized domain for complex scientific computing into a general purpose and widely accessible model for everyday computing. One of the most successful efforts is represented by the use of the multicores available in modern Graphical Processing Units (GPUs) for general purpose parallel computing (General Purpose Graphical Processing Units—GPGPUs). Several libraries and programming environments (e.g., the *Compute Unified Device Architecture* (*CUDA*) created by *NVIDIA*) have been made available to allow programmers to access GPGPUs and exploit their computational power.

In this paper, we propose the design and implementation of a novel *constraint solver* that exploits parallel *Local Search* (*LS*) using GPGPU architectures to solve constraint optimization problems. The optimization process is performed in parallel on multiple large promising regions of the search space, with the aim of improving the quality of the current solution. The local search model pursued is a variant of *Large Neighborhood Search (LNS)* [18, 3]. LNS is a local search techniques characterized by an alternation of *destroy* and *repair* methods—used to destroy and repair part of a solution of a problem. The perturbation of the solution caused by each destroy-repair step is significant, enabling the exploration of a large neighborhood of the search space within each iteration. Each neighborhood is explored using local search strategies and the best neighborhood (i.e, the one that better improves the quality of the solution) is selected to update the variables accordingly. The use of GPGPUs allows us to speed-up this search process and represents an alternative way to enhance performance of constraint solvers. The main contributions of this paper are:

1) Novel design and implementation of a constraint solver performing parallel search. Unlike the traditional approaches to parallelism, we take advantage of the computational power of GPGPUs. These architectures can provide thousands of parallel processing units and they are readily available in the form of graphic cards in most modern computers. To the best of our knowledge this is the first constraint solver system that uses GPGPU to perform parallel local search.

2) A general framework that exploits *Single-Instruction Multiple-Threads* (*SIMT*) parallelism to speed-up local search strategies. We will present six different local search strategies that can be used to explore in parallel multiple large neighborhoods. These strategies are implemented by making very localized changes in the definition of a neighborhood. Hence, the user needs only to specify the structure of a neighborhood, without worrying about how it is actually performed the underlying parallel computation.

3) A hybrid method for solving constraint optimization problems that uses local search strategies on large neighborhoods of variables. Usually, large neighborhood are explored using standard CP techniques. Instead, we present an approach based on local search to find the neighborhood that improves the objective function the most among a large set of different neighborhoods.

2 Background

A *Constraint Satisfaction Problem (CSP)* [14] is defined as $\mathcal{P} = (X, D, C)$ where: **(1)** $X = \langle x_1, \ldots, x_n \rangle$ is an n-tuple of vari-

[1] Dept. Mathematics & Computer Science, University of Udine and Dept. Computer Science, New Mexico State University

ables; **(2)** $D = \langle D^{x_1}, \ldots, D^{x_n} \rangle$ is an n-tuple of domains, each associated to a distinct variable in X, and **(3)** C is a finite set of constraints on variables in X, where a constraint c on the m variables $x_{i_1}, \ldots, x_{i_m}$, denoted as $c(x_{i_1}, \ldots, x_{i_m})$, is a relation $c(x_{i_1}, \ldots, x_{i_m}) \subseteq \times_{j=i_1}^{i_m} D^{x_j}$. A *solution* of a CSP is a tuple $\langle s_1, \ldots, s_n \rangle \in \times_{i=1}^{n} D^{x_i}$ s.t. for each $c(x_{i_1}, \ldots, x_{i_m}) \in C$, we have $\langle s_{i_1}, \ldots, s_{i_m} \rangle \in c(x_{i_1}, \ldots, x_{i_m})$. $\mathcal{P}$ is (in)consistent if it has (no) solutions. A *Constraint Optimization Problem* (*COP*) is a pair $\mathcal{Q} = (\mathcal{P}, g)$ where $\mathcal{P}$ is a CSP, and $g : \times_{i=1}^{n} D^{x_i} \to \mathbb{N}$ is a *cost function*. Given $\mathcal{Q}$, we seek a solution s of $\mathcal{P}$ such that $g(s)$ is minimal (maximal) among all solutions of $\mathcal{P}$.

Typical CSP solvers alternate two steps: **(a)** Selection of a variable and non-deterministic assignment of a value from its domain (*labeling*), and **(b)** Propagation of the assignment through the constraints, to reduce the admissible values of the remaining variables and possibly detect inconsistencies (*constraint propagation*). COP solvers follows the same scheme but they explore the space of possible solution of the problem in order to find the optimal one, e.g., using branch and bound techniques. A COP solver stops whenever exploration is complete or a given limit is reached (e.g., time/number of improving solutions), returning the best solution found so far.

Local Search (LS) techniques [1, 14] deal with COPs and are based on the idea of iteratively improving a candidate solution s by minor "modifications" in order to reach another solution s' from s. The set of allowed modifications is called the *neighborhoods* of s and it is often defined by means of a neighborhood function η applied to s. LS methods rely on the existence of a candidate solution. Most problems typically have a naive (clearly not optimal) solution. If this is not the case, some constraints can be relaxed and a LS method is used with a cost function based on the number of unsatisfied constraint: when a solution of cost 0 is found, it will be used as a starting point for the original CSP. Other techniques (e.g., a constraint solver) might be used to determine the initial candidate solution.

Large Neighborhood Search (LNS) [18, 20] is a technique that hybridizes CP and LS to solve optimization problems. It is a particular case of local search where $\eta(s)$ generates a (random) neighborhood larger than those typically used in LS. The difference is that these sets of candidate solutions are explored using constraint based techniques, and the best improving solution is looked for. If after a timeout an improving solution is not found, a new random neighborhood is attempted. The process iterates until some stop criteria are met. Technically, all constraints among variables are considered, but the effect of $\eta(s)$ is to destroy the assignment for a set of variables. The stop criteria can include a global timeout or a maximum number of consecutive choices of $\eta(s)$ that have not lead to any improvements.

Modern *Graphics Processing Units* (GPUs) are multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy to support graphical processing. In this paper, we consider the *CUDA* programming model proposed by NVIDIA [15], which enables the use of the multiple cores of a graphic card to accelerate general (non-graphical) applications. The underlying model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT),* where the same instruction is executed by different threads that run on identical cores, grouped in *Streaming Multiprocessors* (*SMs*), while data and operands may differ from thread to thread.

A typical CUDA program is a C/C++ program. The functions in the program are distinguished based on whether they are meant for execution on the CPU (referred to as the *host*) or in parallel on the GPU (referred as the *device*). The functions executed on the device are called *kernels*, where each kernel is a function to be executed by several *threads*. To facilitate the mapping of the threads to the data structures being processed, threads are grouped in *blocks*, and have access to several memory levels, each with different properties in terms of speed, organization and capacity. CUDA maps blocks (coarse-grain parallelism) to the SMs for execution. Each SM schedules the threads in a block (fine-grain parallelism) on its computing cores in chunks of 32 threads (*warps*) at a time. Blocks are organized in a 3D *grid*, and hence a kernel is executed by a grid of parallel thread blocks. Threads within a block can communicate by reading and writing a common area of memory (*shared memory*). Communication between blocks and communication between the blocks and the host is realized through a large slow *global memory.*

The development of CUDA programs that efficiently exploit SIMT parallelism is a challenging task. Several factors are critical in gaining performance. Memory levels have significantly different sizes (e.g., registers are in the order of dozens per thread, shared memory is in the order of a few kilobytes per block) and access times, and various optimization techniques are available (e.g., *coalesced* of memory accesses to contiguous locations into a single memory transaction). Thus, optimization of CUDA programs requires a thorough understanding of GPU's hardware characteristics.

3 Solver Design and Implementation

Overall Structure of the Solver: The structure of our constraint solver is based on the general design recently presented in [5]—where a GPU architecture is used to perform parallel constraint propagation within a traditional event-driven constraint propagation engine [16]. We adopt this design to compute a first feasible solution to be successively improved via LNS.[2] Variable's domains are represented using *bit-masks* stored in ℓ *unsigned int* 32-bit variables (for a suitable ℓ), while the status of the computation at every node of the search tree is represented by a vector of bit-masks corresponding to the current domains of all the variables in the model. The supported constraints correspond to the set of finite domain constraints that are available in the *MiniZinc/FlatZinc* modeling language [12]. We modify the `solve` directive of FlatZinc to specify the local search strategy to be used during the neighborhood exploration.

The solver manages two types of variables: **(1)** Standard *Finite Domain* (*FD*) variables and **(2)** *Auxiliary* (*Aux*) variables. Aux variables are introduced to represent FlatZinc intermediate variables and they are used to compute the objective function. Their domains are initially set to all allowed integer values.

We denote with $x^{\text{aux}}_{\text{fobj}}$ the Aux variable that represents the cost of the current solution. The search is driven by assignments of values to the FD variables of the model. The value of Aux variables is instead obtained via constraint propagation.

After a solution s is found, a neighbor is computed using $\eta(s)$, by randomly selecting a set of variables to be "released" (i.e., unassigned). The use of a GPU architecture allows us to concurrently explore several of these sets $\mathcal{N}_1, \ldots, \mathcal{N}_t$, all of them randomly generated by $\eta(s)$. Let m be a fixed constant; we compute m initial assignments for the variables in the set $\mathcal{N}_i$—these are referred to as the *(LS) starting points* $SP_{i,j}$ ($i = 1, \ldots, t$ and $j = 1, \ldots, m$). The starting points can be computed in two ways. In the first option (*random*), each $SP_{i,j}$ is obtained by randomly choosing values from the domains of the variables in $\mathcal{N}_i$. This random assignment might not produce a solution of the constraints. However, for problems with a high number of solutions, this choice can be an effective LNS starting point. In the second option (*CP*), a random choice is performed only

[2] It is also possible to specify an initial solution as input, if known.

for the first variable in $\mathcal{N}_i$; this choice is followed by constraint propagation, in order to reduce the domains of other variables; in turn, a random choice is made for the second variable, using its reduced domain, and so on. If this process leads to a solution, then such solution is used as a starting point $SP_{i,j}$. Otherwise a new attempt is done. It is of course possible to implement other heuristics for the choices of the variables and their values (e.g., *first_fail*, *most constrained*). If the process leads to failure for a given number of consecutive attempts, only the already computed $SP_{i,j}$ (if any) are considered.

A total of $128 \cdot k$ $(1 \leq k \leq 8)$ threads (a block) are associated to each $SP_{i,j}$ belonging to the correspondent set $\mathcal{N}_i$. These threads will perform LS starting from $SP_{i,j}$ (Fig. 1). The value of k depends on the architecture and it is used to split the computation within each starting points, as described in what follows.

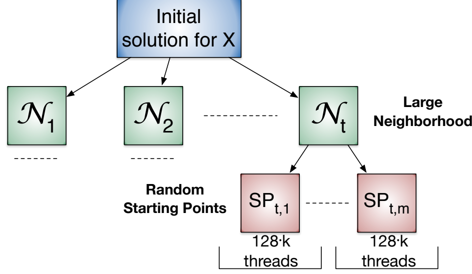

Figure 1: Parallel exploration of subsets $\mathcal{N}_i$ of variables. A LS strategy explores the search space of $\mathcal{N}_i$ in parallel from different starting points SP_{i_j}.

When all the threads end their computations—according to a given LS algorithm (see Sect. 4)—we select among all of them the solution σ that optimizes the value $x^{\text{aux}}_{\text{fobj}}$ among all solutions $\sigma_{i,j}$ computed. This solution is compared with the previous one and, in case, σ is stored as the new best solution found so far.

This process is repeated for h *iterative improving* steps (*II*), each restarting from the best found so far, but changing the randomly generated subsets of variables $\mathcal{N}_i$. After h IIs, the process restarts from the initial solution and is repeated for s *restarts* or until a given time-out limit is reached. In the end, the best solution found during the whole search process is restored. For example, the following directives written in the model cause the solver to select $t = 2$ subsets $\mathcal{N}_i$, each containing 50% of the whole set of variables X (randomly chosen), with $m = 4$ SP per subset, $s = 100$ restarts, and a time-out limit of 600 sec:

```
lns( 50, 2, 4, 10, Gibbs, 100, 600 );
solve minimize fobj;
```

The solver tries to improve the value of $x^{\text{aux}}_{\text{fobj}}$ in $h = 10$ II using *Gibbs* sampling as LS strategy—see Sect. 4.

Exploiting GPU Parallelism. Let us describe more in detail how we divide the workload among parallel blocks, i.e, the mapping between the subsets of variables $\mathcal{N}_i$ and CUDA blocks. The complete set of constraints, including those involving the objective function, and the initial domains are static entities; these are communicated to the GPU once at the beginning of the computation. We refer to the *status* as the current content of the domains of the variables—in particular, an assigned variable has a singleton domain. As soon as the solver finds a feasible solution, we copy the status into the global memory of the device, as well as the t subsets of variables representing the neighborhoods to explore. The CPU is in charge to launch the sequence of kernels with the proper number of blocks and threads. In what follows we focus on a single II step since the process remains the same for each restart s (the CPU stores the best solution found among all restarts). At each iterative improving step r, $0 \leq r \leq h$, the CPU launches the kernel K_1^r with $t \cdot m$ blocks, where each block is assigned its own $SP_{i,j}$. Each block contains $128k$ threads. A kernel of type K_1 starts a local search optimization process from each starting point in order to explore different parts of the search tree at the same time. The current global status will be updated w.r.t. the best neighborhood selected among all.

After the kernel K_1^r has been launched by the host, the control goes back immediately to the CPU which calls a second kernel K_2^r that will start the computation on GPU as soon as K_1^r has finished. This kernel is in charge of performing a parallel reduction on the array of costs computed by K_1^r. It can be the case that in some blocks, the LS strategy implemented is unable to find a solution; in this case the corresponding value is set to $\pm\infty$ (according to minimization/maximization). Moreover, K_2^r updates the status with the new assignment σ of values for the variables in the subsets $\mathcal{N}_i^r$ that has led to the best improvement of $x^{\text{aux}}_{\text{fobj}}$.

At each II, r is incremented. If $r \leq h$ then the CPU will select t new subsets of variables $\mathcal{N}_i^{r+1}$ for the following cycle. Also this operation is performed asynchronously w.r.t. the GPU, i.e., the new subsets of variables are copied to the global memory of the device by a call to an asynchronous `cudaMemcpy` instruction. As a technical note, the array containing the new subsets is allocated on the host using the so-called *pinned* (i.e., host-locked) memory that is necessary in order to perform asynchronous copies between host and device.

When the time limit is reached or $r > h$, host and device are synchronized by a synchronous copy of the current status from the GPU to the CPU (Fig. 2). If the time limit is not reached and another restart has to be performed, the current solution is stored (if it improves the current objective value), the objective function is relaxed, and the whole process is repeated.

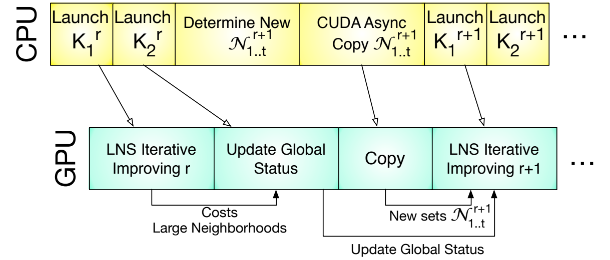

Figure 2: Concurrent computations between host and device.

A portion of the global memory of the GPU is reserved to store the status, the array representing the sets $\mathcal{N}$, and an array of size $(1 + |\mathcal{N}|) \cdot t \cdot m$ of 32 bits unsigned integer, to store the assignment and the correspondent cost for each starting point.

As anticipated above, an additional level of parallelism is exploited using the threads belonging to the same block focused on the LS part (kernel K_1^r). Precisely, K_1^r is launched with $128k$ threads (i.e., $4k$ warps) per block. We use parallelism at the level of warp to avoid divergent computational branches for threads belonging to the same warp. Divergent branches do not fit into the SIMT model, and cause a decrease of the real parallelism achieved by the GPU.

First, all of the threads are used to speed-up the copy of the current status from the global to the shared memory, and to restore the domains of the Aux variables. The queue of constraints to be propagated is partitioned among warps, according to the kind of variables involved: **(1)** FD variables only, **(2)** FD variables and *one* Aux variable, **(3)** two or more Aux variables, and **(4)** $x^{\text{aux}}_{\text{fobj}}$. Since the process starts with $SP_{i,j}$, the constraints of type **(1)** are only used to check consistency when random option for SP is used. This is done us-

ing the first two warps (i.e., threads $0 \dots 64k-1$). Observe that the use of a thread per constraint might lead to divergent computational branches, when threads have to check consistency of different constraints. As soon as a warp finds an inconsistent assignment, it sets the value of the $x^{\text{aux}}_{\text{fobj}}$ variable to $\pm\infty$ in the shared memory, as well as a global flag to inform the other threads to exit. Constraints of type **(2)** propagate information to the unique Aux variable involved. This can be done in parallel by the other two warps (i.e., threads $64k \dots 127k$).

If no failure has been found, all threads are synchronized in order to be ready to propagate constraints of type **(3)**. This propagation phase requires some sequential analysis of a queue of constraints and a fixpoint computation. To reduce the numbers of scans of this queue, we use the following heuristic: we sort the queue in order to consider first the constraints that involve variables that are also present in constraints of type **(2)**, and only later constraints that involve only Aux variables. The idea is that the Aux variables that are present in constraints of type **(2)** are already assigned after their propagation, and hence can propagate to the other Aux variables. We experimentally observed that this heuristic reduces the number of scans to one in most of the benchmarks. We use all warps to propagate this type of constraints. In practice, we divide the queue in $4k$ chunks, and we loop on these chunks until all variables are ground or an inconsistent assignment has been detected. Finally, threads are synchronized again and the value of the variable $x^{\text{aux}}_{\text{fobj}}$ is computed propagating the last type of constraints. (see Fig. 3).

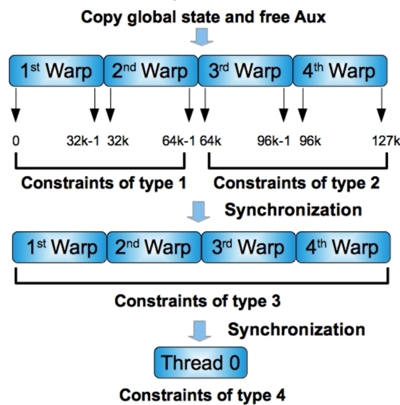

Figure 3: Thread partition within a block.

Some Technical Details. Since the whole process is repeated several times, some FD variables and Aux variables need to be released. This process is done exploiting CUDA parallelism, as well. In our experiments we set $k = 4$, and hence we use 512 threads per blocks for splitting the constraints. The splitting is parametric w.r.t. k A greater (or lower) number of threads is, of course, possible since kernel invocations and splitting are parametric w.r.t. the value k. The reason behind 512 depends on the specific GPU we are using and the number of SMs available. In particular, a larger number of threads would require more resources on the device, leading to a slower context switch between on blocks. Experiments allowed us to observe that for our hardware 512 threads is a good compromise between parallelism and resources allocated to each block. However, this is a compiler parameter that can be changed for other platforms.

We introduced the possibility to select a further level of parallelism based on the size of the domains—suitable to support some of the LS strategies discussed in Sect. 4 (e.g., *ICM*). These strategies may explore the whole domain of a FD variable in order to select the best assignment. This exploration can be done in parallel, by assigning $64k$ threads to the first half of the domain and $64k$ threads to the second half (i.e., the queues of constrains will be splint in $64k$ chunks instead of $128k$).

The design presented so far does not depend on the local search strategy adopted, as long as it guarantees that each variable is assigned a value. We also require that the status does not exceed 49KB, since this is a typical limit for the shared memory of a block in the current GPUs. If the size of the problem is greater than this threshold, we copy chunks of status into the local memory according to the variables involved in the current queue of constraints to propagate.

4 Local Search Strategies

We have implemented six LS strategies for testing our framework. These strategies lead from a solution s to s' by repeatedly applying η on the set $\mathcal{N}$ of variables that can be re-assigned. After the action, constraints consistency is checked and $x^{\text{aux}}_{\text{fobj}}$ is computed. New strategies can be added as long as they implement a function η starting from s and from a subset of variables $\mathcal{N}$. We stress that the primary purpose of the LS presented in this section is to show how these methods can take advantage of the underlying parallel framework, more than the quality of the results they produce. Ad-hoc LS strategies should be implemented based on the problem to solve.

1) The *Random Labeling (RL)* strategy randomly assigns to the variables of $\mathcal{N}$ values drawn from their domains. This strategy might be effective when we consider many sets $\mathcal{N}$, and the COP is not very constrained. It can be repeated a number p of times.

2) The *Random Permutation (RP)* strategy performs a random permutation (e.g., using Knuth's shuffling algorithm [9]) of the values assigned to the variables in $\mathcal{N}$ in s and updates the values according accordingly. It can be used on problems where the domains of the variable are identical (e.g., *TSP*). It can be repeated p times.

3) The *Two-exchange permutation (2P)* strategy swaps the values of one pair of variables in $\mathcal{N}$. The neighborhood size is $n = \frac{|\mathcal{N}|(|\mathcal{N}|+1)}{2}$, and we force the number of starting points to be less than of equal to n.

4) The *Gibbs Sampling (GS)* strategy [4] is a simple *Markov Chain Monte Carlo* algorithm commonly used to solve the maximum a-posteriori estimation problem. We use it for COPs in the following way. Let ν be the current value of $x^{\text{aux}}_{\text{fobj}}$. The function f is defined as follows: *for each* variable x in $\mathcal{N}$, choose a *random* candidate $d \in D^x \setminus \{s(x)\}$; then determine the new value ν' of $x^{\text{aux}}_{\text{fobj}}$, and accept or reject the candidate d with probability $\frac{\nu'}{\nu}$.[3] This process is repeated for p samplings steps; for p large enough, the process converges to the a local optimum for the large neighborhood.

5) The *Iterated Conditional Mode (ICM)* [4] can be seen as a greedy approximation of Gibbs sampling. The idea is to consider one variable $x \in \mathcal{N}$ at the time, and evaluate the cost of the solution for all the assignments of x satisfying the constraints, keeping all the other variables fixed. Then x is assigned with the value that minimize (maximize) the costs. To speed-up this process, all values for x are evaluated in parallel, splitting the domain of D^x between two groups of $2k$ warps each.

6) The *Complete Exploration (CE)* enumerates all the possible combination of values of the variables in $\mathcal{N}$. Given an enumeration $\vec{d_1}, \dots, \vec{d_e}$ of these values, each $\vec{d_i}$ is assigned to a block i, and the corresponding cost function is evaluated. The assignment with the best solution is returned. This method can be adopted when the product of the size of domain's variables of $\mathcal{N}$ is not huge.

[3] Some other details must be considered depending on the minimization/maximization options and positive/negative values.

5 Experiments

We implemented CPU and GPU versions of the LNS-based solver called *CPU/GPU-LNS* respectively. We first compare the two versions of the solver. Then, we compare the GPU-LNS against a pure CP approach in JaCoP [10], and a LNS implementation in *OscaR* [13]. We run our experiments on a CPU AMD Opteron (TM), 2.3GHz, 132 GB memory, Linux 3.7.10-1.16-desktop x86_64, and GPU GeForce GTX TITAN, 14 SMs, 875MHz, 6 GB global memory, CUDA 5.0 with compute capability 3.5. In what follows we report only the most significant results. The interested reader can visit `http://clp.dimi.uniud.it/sw/cp-on-gpu/` for a more extensive set of tests and benchmarks. In all tables t ($|\mathcal{N}|$) denotes the number (size) of large neighbors, m the number of SP per neighbor, times are reported in seconds, and best results are **boldfaced**.

CPU vs GPU: solving CSPs We compared CPU and GPU on randomly generated CSPs defined by $\neq$ constraints between pairs of variables. We use this benchmark to test the performance of GPU-LNS on finding feasible starting points SP (see option CP, Sect. 3). Table 1 reports the results in seconds for a CSP consisting of 70 variables and 200 constraints. In these experiments SP are generated considering one variable at a time, assigning it randomly with a value in its domain and subsequently propagating constraints to reduce domains of the other variables. When the number of $SP_{i,j}$ increases, speedups of one order of magnitude w.r.t. the sequential implementation are obtained. A high number of parallel tasks pays off both the different speed of the GPU cores w.r.t. the CPU cores and the device global memory latencies.

Table 1: CPU vs GPU: solving CSP

$\|\mathcal{N}\|$	t	m	CPU-LNS(s)	GPU-LNS(s)	Speedup
20	1	1	**0.216**	0.218	0.99
20	50	50	1.842	**0.379**	4.86
20	100	100	6.932	**0.802**	8.64
30	1	1	**0.216**	0.218	0.99
30	50	50	2.460	**0.377**	6.52
30	100	100	8.683	**0.820**	10.58

CPU vs GPU: evaluating LS strategies. CPU and GPU solvers have been compared considering the LS strategies of Sect 4. As benchmark we considered a *Mod*ified version of the k-*Coloring Problem* (*MKCP*). The goal is to maximize the difference of colors between adjacent nodes, i.e. $\max \sum_{(i,j)\in E} |x_i - x_j|$, where x_i (x_j) represents the color of the nodes i (j), provided pairs of adjacent nodes are constrained to be different. Here we report the results concerning on one instance[4] of a graph with 67 nodes and 232 edges, that requires 4423 Aux variables and 4655 constraints. The initial solution (value 2098 with domains of size 30) is found by a *leftmost* strategy with increasing value assignment (this time has not been considered in the table). Since in this experiments our goal is just to compare CPU and GPU times, we run tests with the same pseudo-random sequence, $h = 10$ and $s = 0$. Results are reported in Table 2. For the LS and RP we considered $p = 5$ repetitions.[5] Better speedups are in obtained for larger neighborhoods and in particular for the RL method and the CE method which are the less demanding strategies (GPU cores receive simple but numerous task to execute). On the other hand, the higher speedups are obtained by the CE strategy. Using CE we considered only one neighborhood reducing its size to 2, 3 and varying the domains size from 10 to 30.

Table 2: MKCP benchmark using six LS strategies (maximization)

LS	$\|\mathcal{N}\|$	t	m	Min	CPU-LNS(s)	GPU-LNS(s)	Speedup
RL	20	1	1	22828	0.206	0.359	0.57
RL	20	50	50	28676	9.470	0.603	15.70
RL	20	100	100	**29084**	35.22	1.143	30.81
RL	30	1	1	20980	0.218	0.258	0.84
RL	30	50	50	27382	7.733	0.615	12.57
RL	30	100	100	29028	43.24	1.394	**31.01**
RP	20	1	1	15902	0.046	0.069	0.66
RP	20	50	50	17586	13.59	4.154	3.27
RP	20	100	100	**17709**	53.32	16.28	**3.27**
RP	30	1	1	16489	0.045	0.068	0.66
RP	30	50	50	17375	13.49	4.187	3.22
RP	30	100	100	17527	53.88	16.46	3.27
2P	10	1	1	15073	0.151	0.062	2.43
2P	10	20	20	16541	1.231	0.381	3.23
2P	10	50	50	16636	2.839	0.832	**3.41**
2P	20	1	1	15083	0.285	0.119	2.39
2P	20	20	20	16628	4.597	1.351	3.40
2P	20	50	50	**16646**	11.11	3.267	3.40
GS	10	1	1	26486	0.546	1.910	0.28
GS	10	10	10	29308	28.09	12.15	2.31
GS	10	50	50	**30810**	724.2	279.6	2.59
GS	30	1	1	24984	1.053	4.880	0.21
GS	30	10	10	27722	78.59	33.84	2.32
GS	30	50	50	28546	1982	747.92	**2.65**
ICM	5	1	1	31718	0.644	1.637	0.39
ICM	5	10	10	32204	32.23	7.650	4.21
ICM	5	20	20	32296	120.8	26.50	4.55
ICM	20	1	1	31948	0.993	2.522	0.39
ICM	20	10	10	32202	25.55	4.636	5.51
ICM	20	20	100	**32384**	92.68	13.26	**6.98**
CE	2	1	100	8004	0.692	0.324	2.13
CE	3	1	1000	9060	3.932	0.829	4.74
CE	2	1	400	17812	2.673	0.279	9.58
CE	3	1	8000	20020	43.26	1.298	33.32
CE	2	1	900	24474	3.444	0.817	4.21
CE	3	1	27000	**29262**	83.06	2.159	**38.47**

Table 3: Minizinc benchmarks (minimizazion problems, save Knapsack).

System	Benchmark	First Sol	Best Sol(sd)	Time(s)
JaCoP	Transportation	6699	6640	600
JaCoP	TSP	10098	6307	600
JaCoP	Knapsack	7366	15547	600
JaCoP	Coins_grid	20302	19478	600
GPU-LNS	Transporation	7600	**5332** (56)	57.89
GPU-LNS	TSP	13078	**6140** (423)	206.7
GPU-LNS	Knapsack	0	**48219** (82)	6.353
GPU-LNS	Coins_grid	20302	**16910** (0)	600

Table 4: Quadratic Assignment Problem (minimization)

System	q	First Sol	Best Sol (sd)	Time(s)
OscaR	15	79586	9086 (0)	63.09
OscaR	32	430	254 (0)	126.2
OscaR	64	300	212 (0)	1083
GPU-LNS	15	83270	**0** (0)	0.242
GPU-LNS	32	368	**199.6** (9.66)	1.125
GPU-LNS	64	254	**121.6** (2.87)	2.764

Comparison with standard CP In this section we evaluate the performance of the GPU-LNS solver on some Minizinc benchmarks, comparing its results against the solutions found by the state-of-the-art CP solver JaCoP [10]. We present results on medium-size problems which are neither too hard to be solved with standard CP techniques nor too small to make a local search strategy useless.

We considered the following four problems:[6] (1) The *Trasportation* problem, with only 12 variables but the optimal solution is hard to find using CP. The heuristics used for JaCoP is the `first_fail`, `indomain_min`, while for GPU-LNS we used the RL method.

[4] `1-Insertions_4.col` from `http://www.cs.hbg.psu.edu/txn131/graphcoloring.html` Other tests are available on-line.

[5] For the RP strategy we slightly modified the model transforming the coloring benchmark into a permutation problem.

[6] Models and description are available at http://www.hakank.org/minizinc/

We used $t = 100$ neighborhoods of size 3, $m = 100$ SP each, and $h = 500$. (2) The *TSP* with 240 cities and some flow constraints. The heuristics used for JaCoP is the same as above, RP strategy is used in GPU-LNS with $p = 1$. We use $t = 100$ neighborhood of size 40, $m = 100$, and $h = 5000$. (3) The *Knapsack* problem. We considered instances of 100 items[7]. The strategy adopted in JaCoP is `input_order`, `indomain_random`, while for G-LNS we used the RL search strategy, with $t = 50$ neighborhoods of 20 variables, $m = 50$, and $h = 5000$. (4) The *Coins_grid* problem. We considered this problem to test our solver on a *highly* constrained problem. For this benchmark we sightly modified the LS strategy: first we set $\eta(s) = s$, then we used CP (option 2) to generate random SPs. The strategy adopted in JaCoP is `most_constrained`, `indomain_max`, while for G-LNS we used the RL search strategy, with $t = 300$ neighborhoods of 20 variables, $m = 150$, and $h = 50000$. Table 3 reports the first solution value, the best solution found (within 10 min) and the (average on 20 runs for GPU-LNS) running times. For GPU-LNS we also report the standard deviation of the best solution value.

Comparison with Standard LNS. We compared GPU-LNS against a standard implementation of a LNS in *OscaR*. OscaR is a Java toolkit that provides libraries for modelling and solving COP using Constraint Based Local Search [20]. We compare the two solvers on a standard benchmark used to test LNS strategies, namely the *Quadratic Assignment Problem (QAP).*[8] We used three different datasets (small/medium/large sizes). OscaR is run using adaptive LNS with Restart techniques. For each instance we tried different combinations of restarts and adaptive settings; Results for the best combination are reported in 4, as well as GPU-LNS results with the RP strategy, $h = 10$, $t = 50$ neighborhood of size 20, and $m = 50$. For both systems results are averaged on 20 runs and standard deviation of best results is reported. Standard deviations of best solutions are reported. The GPU-LNS version of the solver outperforms OscaR (this is mainly due to the fact that GPU-LNS considers 2500 neighborhoods at a time). We also tried to compare GPU-LNS against OscaR on the *Coins_problem* benchmark. We started both the LNSs from the same initial solution found by OscaR (i.e., 123460), and we used the same setting described above for GPU-LNS. Both system reached the time-out limit with an objective value of 25036 for GPU-LNS, and 123262 for OscaR.

6 Related Work and Conclusion

Extensive research has been conducted focusing on LS and LNS to solve COPs, considering many different variants (see [7] for a survey). While extensive research has also been conducted focusing on parallel constraint solving [8], the use of GPGPUs in CP has been less investigated. The only reported effort is [5] where a constraint solver that performs parallel constraint propagation using GPU is presented. The use of GPUs architectures is not new for speeding up LS strategies. For example, a guideline for design and implementation of LS strategies on GPUs is presented in [21], and in [22].

Motivated by the highly parallel hardware platforms available to the broad community, we presented the design of a constraint solver that performs parallel LNS on GPGPUs architectures. Large neighborhoods are explored using LS techniques with the goal of improving the current solution evaluating a large set of neighborhoods at a time. The choice of local search strategies is twofold: first, incomplete but fast methods are usually preferred for optimization problems where the search space is very large but not highly constrained. Second, with very few changes, the parallel framework adopted for a local search method can be easily generalized to be suitable for many different local search strategies, requiring minimal parameter tuning. Our experimental results show that the solver implemented on GPU outperforms its sequential version. Good results are also obtained by comparing the solver against standard CP and LNS implemented using Moreover, we showed that many LS strategies can be encoded on our framework by changing few parameters, without worrying about how it is actually performed the underlying parallel computation. As future work we plain to exploit a deeper integration within LNS and constraint propagation. The framework should be general enough to allow the user to combine these kernels in order to design any search strategy, in a transparent way w.r.t. the underlying parallel computation. Combining kernels to define different (local) search strategies should be done using a declarative approach, i.e., we plan to extend the MiniZinc language to support the above features.

[7] An hard instance has been generated using the generator that can be found at `http://www.diku.dk/~pisinger/generator`.

[8] The description of the problem and the model used for OscaR are available at https://bitbucket.org/oscarlib/oscar/wiki/lns.

REFERENCES

[1] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* John Wiley & Sons, Chichester (UK), 1997.

[2] A. Aggoun et al. Integrating Rule-based Modeling and CP for Solving Industrial Packing Problems. ERCIM News, 81, 2010.

[3] R.K. Ahuja et al. A Survey of Very Large Scale Neighborhood Search Techniques. *Discrete Applied Math*, 123, 2002.

[4] C. M. Bishop, Pattern Recognition and Machine Learning (Information Science and Statistics) Springer, 2006.

[5] F. Campeotto et al. Exploring the Use of GPUs in Constraint Solving. Proc of PADL, LNCS 8324, pp. 152–167, 2014.

[6] A. Caprara et al., Algorithms for railway crew management. Math. Programming 79:127–141, 1997.

[7] F. Focacci et al. *Local Search and Constraint Programming*. Proc of MIC, pp. 451–454, 2001

[8] I. P. Gent, et al., *A Preliminary Review of Literature on Parallel Constraint Solving*. Proc of Workshop on Parallel Methods for Constraint Solving, 2011.

[9] D. E. Knuth. Seminumerical Algorithms. The Art of Computer Programming 2. Addison-Wesley, pp. 124-125, 1969.

[10] K. Kuchcinski and R. Szymanek. JaCoP Library User's Guide, 2012. `http://jacop.osolpro.com/`.

[11] D. Kurlander et al. Commercial Applications of Constraint Programming. Proc of CP, Springer-Verlag, 1994.

[12] N. Nethercote et al. MiniZinc: Towards a Standard CP Modelling Language. Proc of CP, pp. 529–543, 2007. `www.minizinc.org`.

[13] OscaR Team. *OscaR: Scala in OR*, 2012 Available from `https://bitbucket.org/oscarlib/oscar`.

[14] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier, 2006.

[15] J. Sanders and E. Kandrot. CUDA by Example. An introduction to General-Purpose GPU Programming. Addison Wesley, 2010.

[16] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *ACM TOPLAS*, 31(1), 2008.

[17] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and Programming with Gecode, 2013. Web site: `http://www.gecode.org`.

[18] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. Proc of CP, pp. 417–431, 1998.

[19] H. Simonis. Building Industrial Applications with Constraint Programming. Proc of CCL, Springer-Verlag, 2002.

[20] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[21] T. Van Luong et al. *GPU Computing for Parallel Local Search Metaheuristic Algorithms*. IEEE Trans. Computers, 62(1):173–185, 2013.

[22] T. Van Luong et al. *Large Neighborhood Local Search Optimization on GPUs*. Proc of Workshop on Large-Scale Parallel Processing, 2010.