Towards a GPU-Based Parallel Constraint Solver

F. Campeotto[1,2], A. Dal Palù[3], A. Dovier[1], F. Fioretto[1,2], and E. Pontelli[2]

[1] Dept. Mathematics & Computer Science, Univ. of Udine
[2] Dept. Computer Science, New Mexico State Univ.
[3] Dept. Mathematics, Univ. of Parma

Abstract. Constraint programming has gained prominence as an effective and declarative paradigm for modeling and solving complex combinatorial problems. 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. This paper presents an ongoing work for the development of a constraint solver which exploits GPU parallelism. The work is based on two previous results where constraint propagation and approximated search have been parallelized [5, 6]. We summarize these result and discuss the features we have planned to carry on.

1 Introduction

General Purpose Graphics Processing Units (GPUs) are widely available in common desktop and laptop computers. The computational capabilities of such devices can be exploited to perform general (i.e., not graphic-related) computation. This is possible thanks to C-like programming languages, such as the programming platform CUDA offered and supported by NVIDIA. Graphic cards offers several hundreds parallel cores; however exploiting their parallelism in full is not immediate for all the applications. Several factors are critical in gaining performance and often there is no simple and direct translation from traditional parallel encodings since GPUs support exclusively Single Instruction Multiple Threads (SIMT) parallelism.

This paper presents an ongoing work for the development of a constraint solver which exploits CUDA-GPU parallelism. The solver is referred to as NVIDIOSO, which stands for NVIDIa-based cOnstraint SOlver.

In [5] we presented a first prototype of the solver, where the control of the search is delegated to the CPU while an *event-driven* [18] constraint propagation procedure is parallelized using GPU. Parallelism is exploited by either delegating constraints to various GPU kernels or by parallelizing the propagation activities within a single constraint. We observed speedup for problem instances exhibiting a high number of—even simple—constraints, or when global constraints were considered. Search strategies in [5] were rather naive: the "leftmost" variable is chosen and the "min"imum value is attempted first. Subsequently, we integrate

the use of GPU to explore the large neighborhoods of a given solution, aimed at improving a given cost function [6]. The neighborhoods are explored through sampling-based approaches, e.g., Monte Carlo, Gibbs. We observed significant speedup, which where due to that (1) the local search was delegated exclusively to the GPU, and (2) the SIMT parallelism is suited to approximated search strategy—similar computations are executed in different threads with slight changes in the data addressed. The proposed approach is parametrical with respect to the implemented technique for exploring the neighborhood.

The final step towards the realization of the solver NVIDIOSO involves the development of a set of complete and incomplete search strategies and search techniques. In all options (e.g., choice of the variable with most-constrained/min-domain heuristics, values selection) we will exploit the parallelism offered by GPGPU. For instance, lookahead stages of k consecutive assignments could be attempted in parallel. This information could hence be used to evaluate the most promising one, i.e., to be attempted as first.

The paper is organized as follows: after some background on constraint programming, local search, and GPU in Section 2, we report on the current state of the solver NVIDIOSO, in particular summarizing the contributions presented in [5] and [6] (Section 3). Some related work is discussed in Section 4. Future work and Conclusions end the paper.

2 Background

2.1 Constraint Programming

A *Constraint Satisfaction Problem (CSP)* [16] is a triple (X, D, C) where: $X = \langle x_1, \ldots, x_n \rangle$ is a n-tuple of variables, $D = \langle D^{x_1}, \ldots, D^{x_n} \rangle$ is a n-tuple of finite domains, each associated to a distinct variable in X, and C is a finite set of constraints on variables in X. A constraint $c(x_{i_1}, \ldots, x_{i_m})$ is a subset of the cartesian product $\times_{j=i_1}^{i_m} D^{x_j}$. The variables $x_{i_1}, \ldots, x_{i_m}$ are referred to as the *scope* of c (denoted by $scp(c)$). We assume each $D^{x_i} \subseteq \mathbb{N}$; $\min D^x$ and $\max D^x$ denote the minimum and maximum element of D^x, respectively. 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 $\langle s_{i_1}, \ldots, s_{i_m} \rangle \in c$. A CSP is (in)consistent if it has (no) solutions.

CSP solvers (e.g., Algorithm **??**) alternate two steps: (1) Selection of a variable and non-deterministic assignment of a value from its domain (*labeling*), and (2) Propagation of the assignment through the constraints, to reduce the admissible values of the variables and possibly detect inconsistencies (*constraint propagation*). Thus, at the core of a CSP solver there is a constraint propagation engine, that repeatedly propagates information based on the available constraints. In turn, each constraint $c \in C$ is implemented by a set of propagators $\text{prop}(c)$ acting on the domains of the variables in $\text{scp}(c)$. We refer to $\mathcal{F}$ as the set of all propagators involved in a given CSP instance. More formally, given two n-tuples of domains D_1 and D_2, we say that $D_1 \sqsubseteq D_2$ if, $\forall x \in X$, we have that $D_1^x \subseteq D_2^x$. A *propagator* f is a monotonically decreasing function:

$f(D) \sqsubseteq D$ and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. If $f(D) = D$ for all $f \in \mathcal{F}$, then D is a *fixpoint* of $\mathcal{F}$. A *propagation solver* **i-solv** for a set of propagators $\mathcal{F}$ and an initial domain D finds the greatest domain $D' \subseteq D$ which is a fixpoint of $\mathcal{F}$. **i-solv** start its computation from a subset $F_0 \subseteq \mathcal{F}$ of propagators and the current domains that will be, in general, reduced.

A *Constraint Optimization Problem* (COP) is a CSP where each solution is scored according to a given *cost* function. The role of COP solvers is to find a cost optimal solution. Their resolution process is similar of that of CSP solvers, but in addition they rank each solution according to its cost.

2.2 Local search

Real-world combinatorial optimization problems often require a resolution process which is prohibitively intensive in terms of computational resources. Thus, incomplete search strategies are often preferred to exact approaches. *Local Search (LS)* techniques [1, 16] are incomplete search approaches which deal with COPs by iteratively improving a candidate solution s through slight "modifications". 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. More formally, given a problem Π and an instance $\pi \in \Pi$, the *search space* $S(\pi)$ of instance π is the finite set of candidate solutions $s \in S$. A *neighborhood function* $\eta : S(\pi) \rightarrow S(\pi)$ is the function that determines the position that can be reached in one search step at any given time during the search process.

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 COP. Other techniques (e.g., a constraint solver) might be used to determine the initial candidate solution.

Large Neighborhood Search (LNS) [20, 11] is a technique that hybridizes CP and LS to solve optimization problems. It is a particular case of local search where η defines neighborhoods which are larger than those typically adopted by LS. Differently from LN, in LNS the sets of candidate solutions are explored using a constraint based technique. If after a timeout an improving solution is not found, a new random neighborhood is attempted. The process iterates until some stop criteria is met. Technically, all constraints among variables are considered, but the effect of η is to destroy the assignment for a set of variables. Typical stopping criteria rely on a timeout or on a maximum number of consecutive choices of η which do not lead to any improvements.

2.3 GPUs

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 *Compute Unified Device Architecture* (CUDA) programming model proposed by NVIDIA [17], 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* (*SM*s), while data and operands may differ from thread to thread. CUDA's architectural model is summarized in Figure 1.

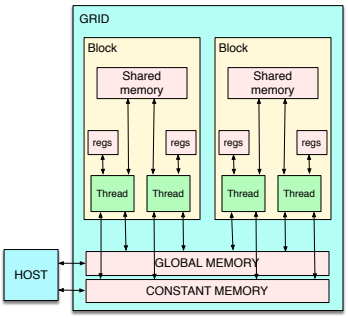

Fig. 1: CUDA Logical Architecture.

A typical CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the *host*) and parts meant for parallel execution on the GPU (referred as the *device*). A parallel computation is described by a collection of *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 *block*, and have access to several memory levels, each with different properties in terms of speed, organization and capacity. Each thread stores its private variables in very fast registers. Threads within a block can communicate by reading and writing a common area of memory (called *shared memory*). Communication between blocks and communication between blocks and the host (i.e., the CPU) is realized through a large (but slow) global memory.

While it is relatively simple to develop correct CUDA programs (e.g., by incrementally modifying a sequential program), it is nevertheless challenging to design an efficient solution. Several factors are critical in gaining performance and often there is no simple and direct translation from traditional parallel encodings. 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., accesses to consecutive global memory locations by contiguous threads can be *coalesced* into a single memory transaction). Thus, optimization of CUDA programs requires a thorough understanding of GPU hardware characteristics.

3 Constraint Programming on GPUs

We present how we have exploited GPUs in the implementation of parts of the Constraint Solver NVIDIOSO. In particular, we focus on the parallelization of the constraint propagation (Sect. 3.1) and on the parallelization of approximated search (Sect. 3.2).

3.1 Parallel Constraint Propagation

Constraint propagators are the core of CSP solvers. Their function is regulated by a *Constraint Propagator* engine, which repeatedly propagates information based on the available constraints, until no more information can be used to further prune the variable's domains. For this purpose, two general decisions take place: (i) which propagators should execute, and (ii) in which order they should execute. These decisions are based on the notion of *event*—a change in the domain of a variable [18].

The engine of NVIDIOSO mixes CPU-based constraint propagation and GPU-based constraint propagation, being able to select the appropriate execution method for each constraint propagator. We base our engine on different levels of parallelism:

Constraint Level: Each constraint $c \in C$ is processed via an independent parallel computation. A mapping of each constraint to a block of threads allows us to fully capitalize on the parallel GPU's SMs. Thus each call to the propagator engine is associated to a kernel whose number of blocks equals the number of the active constraints in the constraint queue.

Variable Level: Within each parallel computation, we process parallel domain reduction for the variables involved in the constraint being propagated. Each variable in the constraint scope is handled by a different thread. Such an approach is particularly suitable to handle global constraints (such as *element*, *inverse* or *table* constraints).

Physical Level: To fully exploit device as well as host characteristics, we categorize each propagator to be processed either on the host or on the device. Such categorization is based on propagators computational complexity: constraints with efficient propagators (e.g. involving few variables), are processed on the host, while the others are delegated to the GPU. Reaching a fixed point during the evolution of the propagation, is ensured by exchanges of information between host and device.

The NVIDIOSO constraint propagation engine is described by Algorithm 1, which is invoked after each labeling step. We denote with F_0 the set of propagators involving exclusively the variables touched during the last labeling step. Line 2 splits F_0 into two sets of propagators: $\mathrm{Q}_{\mathrm{host}}$ and $\mathrm{Q}_{\mathrm{dev}}$, representing the propagators that will be executed on the CPU and those that will be executed by the GPU respectively. These splits are established automatically according to constraint execution properties: e.g., complex constraints which are always delegated to the GPU.

At every loop iteration (lines 4–17) the engine determines the propagators to split in the CPU and the GPU constraint queues. If $\mathrm{Q}_{\mathrm{dev}}$ is not empty, a call to the kernel **gpu_propagate** is invoked (line 6) with a number of block equal to the number of propagator in $\mathrm{Q}_{\mathrm{dev}}$ ($|\mathrm{Q}_{\mathrm{dev}}|$, constraint level parallelism), and a number of threads equal to the maximum scope size (T, variable level parallelism). If $\mathrm{Q}_{\mathrm{host}}$ is not empty, sequential propagation is performed by invoking the function **cpu_propagate** (line 12) that performs propagation on the CPU.

Algorithm 1 gpu-i-solv(F_0, D)

```
1: T ← max{|scp(c)| : c ∈ C};
2: ⟨Q_host, Q_dev ⟩ ← split(F_0);
3: while Q_host ∪ Q_dev ≠ ∅ do
4:    if Q_dev ≠ ∅ then
5:       cudaMemcpy(D_dev, D);
6:       gpu_propagate<<< |Q_dev|, T >>> (Q_dev, D_dev);
7:       cudaMemcpy(D', D_dev);
8:       if failed_event then return false; end if
9:    end if
10:   if Q_host ≠ ∅ then
11:      for f ∈ Q_host do
12:         D'' ← cpu_propagate(f, D);
13:         if failed_event then return false; end if
14:      end for
15:   end if
16:   D_aux ← D; D ← D' ∩ D'';
17:   ⟨Q_host, Q_dev ⟩ ← split(D, D_aux, Q_host ∪ Q_dev);
18: end while
19: return true;
```

Algorithm 2 gpu_propagate(Q, D)

```
1: c_id ← Q[blockIdx];
2: (get_propagators[get_type(c_id)])(c_id, D);
```

If both propagations succeed, the domain D given by of the intersection of the domains D' and D'' is used to determine the new sets of propagators that are not at their fix point for D (lines 16–17).

The **gpu_propagate** kernel (Algorithm 2) executes a propagator per block. It uses its block ID to retrieve the ID of the constraint to propagate from Q (line 1). The function **get_propagators** returns a *pointer to the device function* that implements the (set of) propagators for the constraint c indexed by its type **get_type**(c_id). The constraint identifier *c_id* is also used by the propagator to identify the scope and any parameters of the constraint to propagate.

The propagation on the host is similar to that described above. The kernel invocation is replaced by a *for* loop which iterates over all the propagators in $\mathrm{Q}_{\mathrm{host}}$ (lines 12-15).

For implementation details we refer the interested reader to [5].

3.2 Parallel Incomplete Search

NVIDIOSO implements a set of local searches taking advantage of the GPU parallelism. In particular, we have implemented a Large Neighborhood Search scheme, which requires an admissible (non optimal) solution to the considered problem instance, which is computed using the techniques described in Section 3.1.

Let us briefly describe a sequential version of LNS. Given a valid assignment s for the set of COP variables X, the LNS determines a new solution $s' = \eta(s)$

modifying the values of a subset of variables $\mathcal{N} \subseteq X$. The set $\mathcal{N}$ is referred to as *neighborhood* and it is randomly generated. We call *unassigned* the variables in $\mathcal{N}$ which are modified by the operator η. The admissible solutions identified by the neighborhood are the ones potentially explored by a LS strategy (see below).

From this set of potential solutions a new admissible solution is selected. The latter is referred to as *LS starting point* (or SP) and it may be equal to s. SPs all represent solutions of the COP and they can be computed in different ways. In particular, we implemented two strategies. In the first option, 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, a random choice is performed only 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 for a limited number of times, before discarding the starting point. Constraint propagation is performed by a weak form of AC3 running exclusively on the GPU. This represents a non trivial porting of simplified constraint propagation to GPU. In this implementation, after a variable $x \in \mathcal{N}_i$ has been randomly labelled, the queue of constraint involving x is split among parallel threads.[4] Therefore, each thread performs independent and sequential constraint propagation. To reduce the complexity of the data structures stored in the device global memory, and the consequent number of memory accesses, we perform a weak form of constraint propagation: the AC3 loop is executed only for a fixed number of steps.

The method hence iteratively applies a search strategy to select a new admissible solution by assigning values to the variables in the neighborhood $\mathcal{N}$. The new solution s' may improve the cost function and it can be used for subsequent iterations of the method. This process is repeated for h *iterative improving* steps, each of them restarting from the best solution found so far. Note that at each step a new neighborhood is selected. After h iterations, the process restarts from scratch and is repeated for rst *restarts* or until a given time-out limit is reached. In the end, the best solution found during the whole search process is returned.

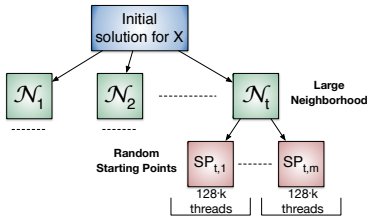

Fig. 2: Parallel exploration of subsets $\mathcal{N}_i$ of variables. A LS strategy explores the search space of $\mathcal{N}_i$ in parallel from different starting points SP_{i_j}.

There are two main aspects where parallelism can help to increase the chances to incur in the optimal solution. Rather than perturbing a single neighborhood

[4] In order to avoid warp divergence the queue of constraints is split among warps of threads instead of assigning different constraints to consecutive threads.

for a solution s, we can perturb t neighborhoods $(\mathcal{N}_1, \ldots, \mathcal{N}_t)$. The GPU parallel paradigm is highly suitable to this aim, as no dependencies occur between each neighborhood. Moreover, rather than computing a single starting point in the neighborhood, m different starting points $SP_{i,j}$ $(i = 1, \ldots, t$ and $j = 1, \ldots, m)$ are retrieved in parallel. This produces $t \times m$ solutions explored in parallel with the selected LS strategy. At the end of the computation, the best solution is retrieved (through an efficient communication between GPU and CPU), compared to s and a new iteration is set up by the CPU.

Let us observe that it is possible to design different parallel frameworks where multiple threads are devoted to propagation for the same CSP associated to a specific LS branch, with the ideas presented above and implemented in a full-GPU version (future work).

From the GPU design point of view, we define $t \times m$ blocks and each of them is in charge of controlling the corresponding starting point $SP_{i,j}$ (Fig. 2). Each block is identified by a group of $128 \cdot k$ $(1 \leq k \leq 8)$ threads, where k depends on the specific GPU in use. These threads handle the operations of the LS algorithm in order to perturb the correspondent starting point $SP_{i,j}$ and hence determine the new solution s' through the function η.

3.3 Local Search Strategies.

We describe six LS strategies, which we will adopt to test our framework. Each strategy follows a similar pattern: the operator η is repeatedly applied on the set $\mathcal{N}$ of *unassigned* variables, so to transit from a solution s to a new solution $s' = \eta(s)$. Let us observe that applying η to s involves performing consistency checking in order to ensure that the assignment s' is a solution. Whereas s' is not a solution we simply set $s' = s$. On the other hand, if s' represent an admissible solution we compute its cost.

1. The *Random Labeling (RL)* strategy randomly assigns to the variables in $\mathcal{N}$ 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 for better chances of finding an admissible solution.
2. The *Random Permutation (RP)* strategy performs a random permutation (e.g., using Knuth's shuffling algorithm [12]) of the values assigned to the variables $\mathcal{N}$ 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 from s of a 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 [3] 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 solution cost. 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 the cost function, and accept or reject the candidate d with probability $\frac{\nu'}{\nu}$. 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)* [3] can be seen as a greedy approximation of Gibbs sampling. The idea is to consider one variable $x \in \mathcal{N}$ at a time, and evaluate the cost of the solution for all the assignments of x satisfying the constraints, keeping all the other variables fixed to their original value in the correspondent SP. Then x is assigned with the value that optimizes the costs.
6. The *Complete Exploration (CE)* enumerates all the possible combinations of values of the variables in $\mathcal{N}$. Given an enumeration $\boldsymbol{d}_1, \ldots, \boldsymbol{d}_e$ of these values, each $\boldsymbol{d}_i$ 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.

3.4 Experiments

We implemented CPU and GPU versions of the constraint solver. 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 experiments are performed on a subset of the local search strategies described above. The interested reader can visit `http://clp.dimi.uniud.it/sw/cp-on-gpu/` for a more extensive set of tests and benchmarks.

Solving CSPs. In [5] we evaluated the performance of our CSP solver w.r.t. the solvers Gecode and JaCoP on some classical benchmarks, specifically *nQueens*, *Schur* (numbers $1, \ldots, N$ in B blocks), and the *propagation stress* benchmarks (see, e.g., the MINIZINC benchmarks folder [15]).

The results showed that our solver is, on average, comparable with the state-of-the-art.

For benchmark problems defined using table constraints (e.g. the *crossword* game, the *Langford* problem, several synthetic problems, and some other real-world problems), a speedup of at least 2 is obtained, showing that the use of the GPU pays off on large instances and real problems.

Solving COPs. In [6] we evaluate the performance of the GPU solver for COPs on some MINIZINC benchmarks,[5] comparing its results against the solutions found by the state-of-the-art CP solver JaCoP [13]. We present results

[5] The interested reader can visit http://clp.dimi.uniud.it/sw/cp-on-gpu/ for a more extensive set of tests and benchmarks.

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 *Transportation* problem with the RL strategy, 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 the GPU implementation we used the RL method.
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 considering the RP strategy. 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 using the RL strategy. We considered instances of 100 items[7]. The strategy adopted in JaCoP is `input_order`, `indomain_random`, while for the GPU version 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 (4) 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 the GPU version we used the RL search strategy, with $t = 300$ neighborhoods of 20 variables, $m = 150$, and $h = 50000$. Table 1 reports the first solution value, the best solution found (within 10 min) and the (average on 20 runs for GPU) running times. For the GPU version we also report the standard deviation of the best solution value. (1) and (2) are minimization problems, while (3) is a maximization problem. Best results are **boldfaced**.

Table 1: MINIZINC benchmarks (minimization 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	Transporation	7600	**5332** (56)	57.89
GPU	TSP	13078	**6140** (423)	206.7
GPU	Knapsack	0	**48219** (82)	6.353
GPU	Coins_grid	20302	**16910** (0)	600

We also compared the GPU solver for COPs 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 [11]. We compare the two solvers on a standard benchmark used to test LNS strategies,

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

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

Table 2: 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	15	83270	**0** (0)	0.242
GPU	32	368	**199.6** (9.66)	1.125
GPU	64	254	**121.6** (2.87)	2.764

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 2, as well as GPU solver 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. Standard deviations of best solutions are reported. The GPU 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 a *highly* constrained benchmark, namely the *Coins_problem*. 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 the GPU implementation. Both system reached the time-out limit with an objective value of 25036 for GPU solver, and 123262 for OscaR.

4 Related Work

Extensive research has been conducted focusing on LS and LNS to solve COPs, considering many different variants (see [9] for a survey). While extensive research has also been conducted focusing on parallel constraint solving [10], the use of GPGPUs in CP has been less investigated. A recent, relevant reference is [2]. The authors implemented the Adaptive Search algorithm, a local search algorithm that can be used for COP and CSP whose search strategy is based on iterative repair. Basically, the degree of unsatisfiability of a constraint is evaluated and helps, together with the cost function in driving the search. A tabu list is also used for avoiding re-visiting recently considered assignments. Our solver is general and their strategy will be implemented in NVIDIOSO as future work. Moreover, NVIDIOSO can run MINIZINC models and deals with constraint propagation using GPU, as well. The solver presented in this paper is an extension of the solver presented in [5] where constraint propagation is performed in parallel using GPGPUs. On the other hand, the use of GPUs architectures is not new for speeding up LS strategies, in particular, for solving

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

hard combinatorial real-world problems. For example, in [19] the authors present an implementation of a LS strategy based on GPU computation and they test it on the vehicle routing problem. In [7] the authors present an implementation of system where local search strategies are performed on GPGPUs considering the protein structure prediction problem. A guideline for design and implementation of LS strategies on GPUs is presented in [14], and in [21]. Instead, in [4] the authors proposes a framework for heterogeneous systems with multiple CPUs and GPUs interconnected through a network. The authors evaluate the framework on the TSP problem and showing its scalability properties considering a system which includes 256 CPUs and 384 GPUs.

Parallelization of SAT solving procedures, which present similar issues at high level, have been also explored [8]. The parallelization of (i) the unit-propagation procedure (performed at every visited node of the search tree) and (ii) the complete parallelization of the "tail" of the search tree have been investigated. The results for (i) indicate that significant speedups arise for large formulas (corresponding to a large number of constraints in a CSP), where the GPU overhead is payed off by parallel work. For (ii) it was observed that parallel expansion of different branches by parallel threads can lead to speedups of one order of magnitude, but it relies on parameter tuning. The main drawback is the divergence of code, namely threads that execute different code, due to different branching in the search algorithm, which degrades the performances.

5 Future Work and Conclusions

We have reported on an ongoing project on developing a constraint solver for finite domains which exploits the power of GPGPU computing. Motivated by the highly parallel hardware platforms available to the broad community, we presented the design of a constraint solver that performs parallel constraint propagation as well as parallel LNS on GPGPUs architectures. Our results show the potential of GPU parallelism in both approaches.

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.

Acknowledgment

This research is partially supported by the National Science Foundation under grants number 1345232 and 0947465, and by the INdAM-GNCS Project 2014.

References

1. E. H. Aarts and J. K. Lenstra. *Local search in combinatorial optimization.* Princeton University Press, 2003.
2. A. Arbelaez and P. Codognet. A GPU Implementation of Parallel Constraint-Based Local Search. In *Parallel, Distributed and Network-Based Processing (PDP)*, pages 648–655, 2014.
3. C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., 2006.
4. M. Burtscher and H. Rabeti. A scalable heterogeneous parallelization framework for iterative local searches. In *IPDPS*, pages 1289–1298, 2013.
5. F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto, and E. Pontelli. Exploring the Use of GPUs in Constraint Solving. In *Practical Aspects of Declarative Languages*, volume 8324 of *Lecture Notes in Computer Science*, pages 152–167, New York, 2014. Springer Verlag.
6. F. Campeotto, A. Dovier, F. Fioretto, and E. Pontelli. GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems. Technical report, TR-CS-NMSU-2014-4-25, NMSU, 2014.
7. F. Campeotto, A. Dovier, and E. Pontelli. Protein Structure Prediction on GPU: a Declarative Approach in a Multi-agent Framework. In *Proc. of 2013 International Conference on Parallel Processing.* IEEE Computer Society, 2013.
8. A. Dal Palù, A. Dovier, A. Formisano, and E. Pontelli. Exploiting Unexploited Computing Resources for Computational Logics. In *Proceedings of the 9th Italian Convention on Computational Logic*, volume 857 of *CEUR Workshop Proceedings*, pages 74–88, Aachen, Germany, 2012. CEUR-WS.org.
9. F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In *Handbook of metaheuristics*, pages 369–403. Springer, 2003.
10. I. P. Gent, C. Jefferson, I. Miguel, N. Moore, P. Nightingale, P. Prosser, and C. Unsworth. A preliminary review of literature on parallel constraint solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving.* Citeseer, 2011.
11. P. V. Hentenryck and L. Michel. *Constraint-based local search.* The MIT Press, 2009.
12. D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms. II.* Addison-Wesley, 1969.
13. K. Kuchcinski and R. Szymanek. Jacop library. users guide, 2009.
14. T. V. Luong, N. Melab, and E.-G. Talbi. Large Neighborhood Local Search Optimization on Graphics Processing Units. In *Workshop on Large-Scale Parallel Processing (LSPP) in Conjunction with the International Parallel & Distributed Processing Symposium (IPDPS)*, Atlanta, États-Unis, 2010.
15. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.
16. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier, 2006.
17. J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.
18. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008.

19. C. Schulz. Efficient local search on the gpuinvestigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing*, 73(1):14–31, 2013.
20. P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Principles and Practice of Constraint ProgrammingCP98*, pages 417–431. Springer, 1998.
21. T. Van Luong, N. Melab, and E. Talbi. Large neighborhood local search optimization on graphics processing units. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.