Exploring the Use of GPUs in Constraint Solving

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. This paper presents an experimental study aimed at assessing the feasibility of parallelizing *constraint propagation*—with particular focus on arc-consistency—using *Graphical Processing Units (GPUs)*. GPUs support a form of data parallelism that appears to be suitable to the type of processing required to cycle through constraints and domain values during consistency checking and propagation. The paper illustrates an implementation of a constraint solver capable of hybrid propagations (i.e., alternating CPU and GPU), and demonstrates the potential for competitiveness against sequential implementations.

1 Introduction

Constraint programming has gained prominence as an effective paradigm for problem modeling and solving, with applications to such diverse domains as scheduling, satisfiability testing, optimization, and verification. A typical *Constraint Satisfaction Problem (CSP)* consists of a set of variables, each taking values from an associated finite domain, along with a set of constraints. The constraints are used to restrict the values that different variables can simultaneously assume. Resolving a CSP consists of determining complete assignments of values to the variables that satisfy all the constraints. Constraint programming is frequently used to address combinatorial problems, which are, in general, NP-hard. Solving CSPs is usually achieved by combining backtracking search with forms of consistency checking, to prune values from the variables' domains that are inconsistent with the constraints. Polynomial time techniques like node, arc, path and bound consistency have been developed for this purpose.

The cost of solving complex CSPs has motivated the exploration of techniques to improve the exploration of the search space; parallelism has been recognized as a strong contender, especially with the wider availability of multicore and cluster platforms. A large body of research has been developed to address parallelization of backtracking search on a variety of parallel and distributed platforms.

The research presented in this paper makes a contribution to the domain of parallel constraint solving, by exploring ways of using *Single-Instruction Multiple-Threads (SIMT)* parallelism to reduce the cost of constraint propagation. The choice of SIMT parallelism has two driving motivations. First of all, it is our

belief that this form of parallelism is suitable to the type of processing that constraints are subjected to during consistency checking. Second, SIMT is the style of parallelism that is natively supported by modern *General Purpose Graphical Processing Units (GPGPUs)*. GPGPUs are massive parallel architectures, that are available in the form of graphic cards in most modern computers; they provide hundreds of computing cores at an affordable cost. Exploiting the parallelism offered by GPUs is not trivial—the cores are often significantly slower than CPU cores, they impose restrictions on branching, and provide a complex memory hierarchy with differences in speed, size, and concurrency of accesses.

The contribution of this paper is a feasibility study that demonstrates the potential for using GPGPUs to speedup a constraint propagation engine, based on the notion of events [23]. We propose a methodology to map constraints, variables, and domain elements to threads running on GPU cores, thus enabling the concurrent analysis of arc and bound-consistency and removal of inconsistent domain values. The methodology is implemented in an experimental solver, and shown to produce performance enhancements even in its simple and unoptimized form. The prototype demonstrates also the strengths and weaknesses of GPU parallelism in constraint solving. This is, to the best of our knowledge, the first study investigating the use of GPGPUs in constraint propagation; this study opens the doors to an alternative way to enhance performance of constraint solvers, through the unexploited computational power offered by GPUs.

2 Background

A *Constraint Satisfaction Problem (CSP)* [19] is defined as $\mathcal{P} = (X, D, C)$ 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. 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.
- 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 $c(x_{i_1}, \ldots, x_{i_m}) \subseteq \times_{j=i_1}^{i_m} D^{x_j}$. The variables $x_{i_1}, \ldots, x_{i_m}$ are referred to as the *scope* of c (denoted by $\mathrm{scp}(c)$).

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 $\langle s_{i_1}, \ldots, s_{i_m} \rangle \in c$. $\mathcal{P}$ is (in)consistent if it has (no) solutions.

CSP solvers (e.g., Algorithm 1) 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; its basic component s a function, from domains to domains, referred to as *propagator* [23]. 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 $D_1 \sqsubseteq D_2$. Each constraint

$c \in C$ is implemented by a set of propagators $\text{prop}(c)$ that operate on the m-tuple of domains of the variables in $\text{scp}(p)$. In the paper we denote by $\mathcal{F}$ the set of all propagators considered. 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 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.

Algorithm 1 search(X, D, C, ℓ)

```
1: if ℓ > |X| then
2:    output D; return true;
3: end if
4: for all d in D^{x_ℓ} do
5:    D' ← ⟨D^{x_1}, ..., D^{x_{ℓ-1}}, {d}, D^{x_{ℓ+1}}, ..., D^{x_{|X|}}⟩;
6:    F_0 ← {prop(c) : c ∈ C ∧ x_ℓ ∈ scp(c)};
7:    if i-solv(F_0, D') ∧ search(X, D', C, ℓ + 1) then
8:       return true;
9:    end if
10: end for
11: return false;
```

The procedure **i-solv** (Algorithm 2) iteratively invokes the propagators until the greatest fixpoint is reached. Two general decisions have to be made in order to reach the fixpoint: *(1)* Which propagators should execute, and *(2)* In which order they should execute. These decisions are based on the notion of *events:* an event is a change in the domain of a variable. We distinguish five types of events: **(1)** ***failed_event***: there is a variable x such that $D'^x = \emptyset$. **(2)** ***empty_event***: no event happened, i.e., $D'^x = D^x$ for all variables considered. **(3)** ***sing_event***: there is a variable x such that $|D'^x| = 1$. **(4)** ***bc_event***: there is a variable x such that $\min D'^x > \min D^x$ or $\max D'^x < \max D^x$. **(5)** ***dmc_event***: there is a variable x such that $D'^x \subset D^x$. These events are used to invoke the necessary propagators only, based on the changes to the variables' domains that occurred.

Algorithm 2 i-solv(Q,D)

```
1: D' ← D;
2: while Q ≠ ∅ do
3:    for all f ∈ Q do
4:       D'' ← f(D);
5:       if failed_event then return false; end if
6:       D ← D'';
7:    end for
8:    Q ← new(Q, D', D'');
9: end while
10: return true;
```

The pseudo code in Algorithm 2 is similar to the well-known *AC3* algorithm (c.f., e.g., [19]): the **while** loop (lines 2–9) propagates the constraints in the queue of propagators Q until no changes happen in the domains, i.e., D is a fixpoint for the propagators invoked, or some domain is empty. The procedure

new(Q, D', D'') chooses the new propagators to be inserted in the queue, based on the changes between the original domain D' and the final domain D'' and on the propagators already in Q. As a side-effect, the procedure modifies the values of the calling domain variable in the **search** procedure.

3 GPU computing

Modern graphic cards (*Graphics Processing Units*) are multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy for graphical processing (e.g., DirectX and OpenGL). Efforts like NVIDIA's *CUDA—Compute Unified Device Architecture* [21] aim at enabling the use of the multicores of a GPU to accelerate general applications—by providing programming models and APIs that enable the full programmability of the GPU. In this paper, we consider the CUDA programming model. The underlying conceptual model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT)*, a variant of the popular SIMD model. In SIMT, the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread. CUDA's architectural model is represented in Figure 2.

Different NVIDIA GPUs provide different numbers of cores, organized in a different way, and with different amounts of memory. The GPU consists of a series of *Streaming MultiProcessors (SMs)*; the number of SMs depends on the specific characteristics of each class of GPU—e.g., the Fermi architecture provides 16 SMs. In turn, each SM contains a collection of computing cores (containing a fully pipelined ALU and floating-point unit); the number of cores per SM may range from 8 (in the older G80 platforms) to 32 (e.g., in the Fermi platforms). Each GPU provides access to on-chip memory (for thread registers and shared memory) and off-chip memory (L2 cache, global memory and constant memory)—see Fig. 2.

A logical view of computations is introduced by CUDA, in order to define abstract parallel work and to schedule it among different hardware configurations. 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 to as the *device*). A parallel computation is described by a collection of *kernels*—each kernel is a function to be executed by several threads. Threads spawned on the device to execute a kernel are hierarchically organized to facilitate the mapping of the threads to the (possibly multi-dimensional) data structures being processed: threads are organized in a 3-dimensional structure (called *block*), and blocks themselves are organized in 2-dimensional tables (called *grids*). CUDA maps blocks (coarse-grain parallelism) on the SMs for execution; each SM schedules the threads

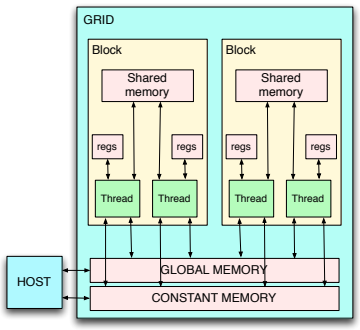

Fig. 2: CUDA Architecture

in a block (fine-grain parallelism) on its computing cores in chunks of 32 threads at a time (called *warps*), thus allowing a group of threads in a block to use the computing resources while other threads of the same block might be waiting for information (e.g., completing a slow memory request). Threads have access to several memory levels, each with different properties in terms of speed, organization (e.g., banks that can be concurrently accessed) and capacity. Each thread stores its private variables in very fast registers (anywhere from 8K to 64K per SM); threads within a block can communicate by reading and writing a common area of memory (called *shared memory*). On the other side, communication between blocks is not supported and it can be accomplished after the completion of the whole kernel. Nevertheless, global memory (up to several GBytes) can be used to store information that can be used by subsequent kernels.

The kernel, invoked by the host, is executed by the device and it is written in standard C-code. The number of running blocks (gridDim) and the number of threads of each block (blockDim) is specified by the kernel call that is invoked on the host code with the following syntax:

$$\mathsf{Kernel} \lll \mathsf{gridDim},\ \mathsf{blockDim} \ggg (\mathsf{param}_1,\ \ldots,\ \mathsf{param}_n);$$

In order to perform a computation on the GPU, it is possible to move data between the host memory and the device memory. By using the specific identifier of each block (blockIdx—providing x, y coordinates of the block in the grid), its dimension (blockDim) and the identifier of each thread (threadIdx—providing x, y, z coordinates for the thread within the block), it is possible to differentiate the data accessed by each thread and code to be executed. For example, the following code fragment shows a kernel and the corresponding call from the host. Each element of a two dimensional matrix is squared, and each thread is in charge of one element of the matrix. The matrix A is represented by a pointer in the device's global memory; CUDA provides functions (e.g., cudaMemCopy) to transfer data between the host and the device's global memory.

```
int main() {   ...                    __global__ sqMatrix(float *Mat){
    dim3 thrsBlock(n,n);                   int i=threadIdx.x;
    sqMatrix<<<1,thrsBlock>>>(A);          int j=threadIdx.y;
    ...                                    Mat[i][j] = Mat[i][j]*Mat[i][j]; }
```

While it is relatively simple to develop correct CUDA programs (e.g., by incrementally modifying an existing sequential program), it is challenging to design an efficient solution. Several factors are critical in gaining performance. The SIMT model requires active threads in a warp to execute the same instruction—thus, diverging flow paths among threads may reduce the amount of actual concurrency. Memory levels have significantly different sizes (e.g., registers are in the order of dozens per thread, while shared memory is in the order of a few kilobytes per block) and access times; different cache behaviors are applied to different memory levels (e.g., constant memory is a cached read-only global memory) and various optimization techniques are used (e.g., accesses to consecutive global memory locations by contiguous threads can be *coalesced* into

a single memory transaction). Thus, optimization of CUDA programs require a thorough understanding of the hardware characteristics of the GPU being used.

4 Parallelizing the Constraint Engine

In this section we describe our approach to GPU-based execution of the **i-solv** procedure presented in Section 2. The corresponding pseudo-code is reported in Algorithm 3.

Our model encodes three different types of parallelism for constraint propagation. Recall that constraint propagation is monotonic, therefore the order in which the data is analyzed does not influence the result (while it might affect the number of operations performed to reach the fixpoint).

`Constraints`: Given a set C of constraints for which propagation and consistency checks are to be performed, a natural form of parallelism is to delegate the processing of each constraint $c \in C$ to a different parallel computation. In particular, it is convenient to map a block of threads (B_c) to the handling of each c, in order to exploit the various parallel GPU's SM.
A kernel with a number of blocks of the size of the current constraint queue C is invoked. Up to 2^{32} blocks can be used on NVIDIA 2.x cards, which is adequate for most CSPs.

`Variables`: A second level of parallelism is applied to the processing of a constraint c assigned to a block B_c. Domain reductions for the variables involved in the constraint (namely $x \in scp(c)$) can be performed in parallel fashion. In particular, each variable can be handled by a different thread that executes the domain filtering. Moreover, the type of operations is executed in a SIMT fashion, since the code for propagation usually repeats identically for each variable. This level of parallelization is suitable to global constraints, such as *element*, *inverse*, or *table* constraint—while it would not bring benefit to constraints that admit efficient propagation algorithms.

`CPU and GPU`: Host and device are capable of independent and parallel work, that can be synchronized by specific programming constructs. We designed a third level of parallelism for constraint propagation, by partitioning the set of propagators in two queues: one to be processed by the CPU and another one by the GPU. Constraints with efficient propagators (e.g. few variables), remains on the host, while the others are delegated to the GPU. During the evolution of the propagation, exchanges of information between host and device ensure to reach the fixpoint faster.

Let us describe the main components of Algorithm 3. At each invocation of the **i-solv** procedure, the set of initial propagators F_0 is split between host and device by the function **split** that initializes the queues of constraints $\mathrm{Q}_{\mathrm{host}}$ and $\mathrm{Q}_{\mathrm{dev}}$ (host and device constraints), based on the type of constraints to be propagated in line 2. The default distribution, based uniquely on the type, can be changed by the **split** function according to two internal thresholds: **(1)** If the number of CPU-propagators is higher than a given *upper bound*, they are

Algorithm 3 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(props(D, D_aux, Q_host ∪ Q_dev));
18: end while
19: return true;
```

all moved to $\mathrm{Q}_{\mathrm{dev}}$; **(2)** If the number of GPU-propagators is lower than a given *lower bound*, they are moved to $\mathrm{Q}_{\mathrm{host}}$.

By varying these bounds, it is possible to force the computation completely on the CPU (huge lower bound) or completely on the GPU (upper bound = 0). These bounds are used to handle the cases where a large number of efficient propagators are assigned to the CPU, while they could take advantage of parallel propagation or, vice-versa, very few expensive propagators are assigned to the GPU, where the time required by memory transactions between host and device would likely offset the advantages of a parallel propagation. The only exception to these rules is for complex constraints (such as the *table* constraint) that are always delegated to the GPU.

Every loop iteration analyzes and modifies the propagators in $\mathrm{Q}_{\mathrm{host}}$ and in $\mathrm{Q}_{\mathrm{dev}}$. If $\mathrm{Q}_{\mathrm{dev}}$ is not empty, parallel propagation is performed by invoking the kernel **gpu_propagate** (line 6), with as many blocks as the size of $\mathrm{Q}_{\mathrm{dev}}$, and as many threads per block as the maximum scope size among all constraints. The kernel function **gpu_propagate** is sketched in Algorithm 4 and explained later. If $\mathrm{Q}_{\mathrm{host}}$ is not empty sequential propagation is performed by invoking the function **cpu_propagate** (line 12). If both propagations succeed, the new states D' and D'', produced respectively by the GPU and the CPU, are merged (line 16) and the function **props**() determines the minimal sets of propagators that are not at their fixpoint for the domain D (line 17). The function **props**() is based on the notion of *events*. It calculates the events based on status D_{aux} of the previous iteration and the current status D ($evts(D, D_{\mathrm{aux}})$), and updates the queue of propagators accordingly:

$$\mathbf{props}(D, D_{\mathrm{aux}}, Q) = \{f \in \mathcal{F} : evt\_set(f) \cap evts(D, D_{\mathrm{aux}}) \neq \emptyset\} \setminus \mathtt{fix}(Q, D)$$

where the set $evt\_set(f)$ is the set of events related to the propagator f, and

$\texttt{fix}(Q, D) = \{f \in Q : f(D) = D\}$. This set of events is computed by analyzing the differences between D and D_{aux}.

Algorithm 4 gpu_propagate(Q, D)

1: $c\_id \leftarrow Q[blockIdx]$;
2: **get_propagators**[**get_type**(c_id)](c_id, D);

Let us briefly discuss Algorithm 4. This kernel invokes a propagator per block. The identifier of the block (*blockIdx*) is used as index on the queue Q to retrieve the identifier *c_id* of the constraint to propagate. 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.[4] A **failed_event** is generated when there is an empty domain. If this is the case, then the propagation will fail and the **i-solve** procedure will return **false**; this will cause the search to backtrack (line 9).

The propagation on the host is similar; the kernel invocation is replaced by a *for* loop that iterates over all the propagators in Q_{host} (lines 12-15). Let us note that, differently from the propagation on the device, the **failed_event** is checked every time a propagator has been considered. Let us discuss some details related to the CPU and GPU implementations of these algorithms.

Domain representation. Domains are represented using *bit-masks* stored in k *unsigned int* 32-bit variables. Precisely, considering $D \subseteq \{0, \ldots, 32k-1\}$ viewing the k variables as a unique string, the domain D is represented by $\sum_{i=0}^{32k-1} 2^i b_i$, where if $i \in D$ then $b_i = 1$, else $b_i = 0$. Negative numbers can be implemented using an appropriate offset value. The use of bit-wise operators on domains reduces the differences between the GPU cores and the CPU cores, since access to data in the former is much slower than in the latter. Three extra variables are used: two for storing the domain bounds ($\min D$ and $\max D$) and one for storing the current event associated to D. We denote with $M = k+3$ the number of variables used. For instance, for storing domains included in [0..927] we use $M = 32$ unsigned int variables.

Status representation. The status of the computation at every node of the search tree is represented by a vector of $M \cdot |V|$ where M is as described above. This representation of the status reduces the total number of accesses to the global memory, since every consecutive 32 domain values are grouped together in a single *integer* value. The choice of M as a multiple of 32 *integers* allows us to take advantage of the device cache, since global memory accesses are cached and served as part of 128-byte memory transactions. Moreover, using the same array of data for both the bit-mask and the domain bounds increases the *coalesced* memory accesses, i.e., the accesses to the global memory are coalesced for contiguous locations in global memory, increasing access performance.

[4] The relationships between constraints and variables (constraint graph) is stored in the device memory, to limit the information exchange between CPU and GPU.

Data transfers. The memory dataflow is designed in order to optimize memory throughput. Since applications should strive to minimize data transfers between the host and the device (i.e., data transfers with low bandwidth), at each parallel propagation step we transfer the minimum information needed to represent the current state in the search tree. Namely, we copy into the global memory of the GPU the previous decisions performed in the current exploration of the search tree, and only the domains of the variables not labeled yet. These domains still ensure a correct execution of the propagation algorithm, as we are interested in reducing only the domains of the variables that are still to be labeled. In order to allow concurrent computations on the host and the device, every **cudaMemcpy** is performed as an asynchronous data transfer. A call to the CUDA function **cudaDeviceSynchronize()**, used to synchronize the host and the device, is requested only when the CPU has finished its sequential propagation.

MiniZinc constraints encoding. In this work we considered the finite domain constraints that are available in the *MiniZinc/FlatZinc* modeling language [15]. Given a MiniZinc model, we translate it and produce an input for our solver in three steps: *(1)* first, we read the MiniZinc file to identify the global constraints being used; *(2)* we translate the model into a FlatZinc model without considering the global constraints (we use the compiler available in the MiniZinc distribution [15]); and *(3)* the FlatZinc translation is given as input to a parser that produces the input for the solver.

Propagators. We have implemented the propagators for the FlatZinc constraints plus specific propagators for some *global* constraints that take advantage of GPU parallelism. As described earlier, every propagator is implemented as a specific device function invoked by a single block. For example, let us consider an *all_different* constraint c on the variables $x_1, \ldots, x_n$, naively encoded as a quadratic number of binary $\neq$ constraints. It can be implemented by a set of n propagators $p_1, \ldots, p_n$, such that the propagator p_i takes care of the constraints $x_i \neq x_j$ where $j \neq i$ (see Algorithm 5). The propagator is typically activated for one i at a time. A sequential implementation of this propagator requires time $\mathcal{O}(n)$, while the parallel version requires $\mathcal{O}(1)$.

Algorithm 5 $p_i(c\_id, D)$

1: $x_i \leftarrow$ **scp**(c_id)[i];
2: $label \leftarrow min\ D^{x_i}$; {$min\ D^{x_i} = max\ D^{x_i}$ since x_i is the current labeled variable}
3: $n \leftarrow$ **scp(**c_id**).size**(); {Constraints information on device global memory}
4: **if** $threadIdx < n \wedge threadIdx \neq i$ **then**
5: $\quad temp \leftarrow$ **scp**(c_id)[$threadIdx$];
6: $\quad D^{temp}[label] \leftarrow 0$;
7: **end if**

Some other constraints require more than one block to fully exploit the parallel computation. This is the case, for example, of the *table* constraint (see Sec. 5). To handle these cases, we modified Algorithm 3 in order to further split the queue $\mathrm{Q}_{\mathrm{dev}}$ in two queues: one for constraints that are propagated using one block per propagator, and one for constraints that use more than one block.

5 Results

We experimentally evaluated our solver using several classical benchmarks. Benchmarks are encoded in MiniZinc and compiled automatically in the solver. In particular, we compare the performance of our solver (in terms of execution time) with that of two state-of-the-art solvers, namely *Gecode* [24] and *JaCoP* [10]. Our solver does not include advanced search strategies at this time—therefore, for a fair comparison, we use Gecode and JaCoP with a naive "leftmost" strategy with increasing value assignment. In order to measure parallel performance, we analyze the speed-ups and limitations of the GPU version against a purely CPU execution of our code—as mentioned earlier, this can be realized by modifying the bounds used to manage the constraint queues. Thus, while the first set of comparisons gives us an idea about the baseline performance of our core solver (including an indication of the overhead introduced to support parallelism), the second set of data measures the improvements gained by using parallelism. We have aimed at creating a core solver that is efficient and competitive with the state-of-the-art, containing overhead to the minimum. All tests have been performed on the following hardware: the *Host* is an AMD Opteron 270, 2.01GHz, RAM 4GB, while the *Device* is an NVIDIA GeForce GTS 450, 192 cores (4MP), Processor Clock 1.566GHz, OS Linux.

Comparison with Gecode and JaCoP. We start by evaluating the performance of our 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]). Let us remark that the *all_different* constraints is implemented in a "quadratic way" in all these problem instances—this explains the relatively slow running times for nQueens. As expected, there are instances that better fit one solver, and other instances that better fit others (see Table 1—running times in seconds). We report two columns for our solver (CPU and GPU). For this experiment, let us focus on the GPU column (the CPU column is used in the following experiments). or a fair comparison, we modified the hybrid and adaptive recomputation parameters of Gecode. In particular we switched off cloning by setting the value c_d (*commit distance*) greater than the expected depth of the search tree.

The labeling strategy for our solver, Gecode and JaCoP is the naive "leftmost" strategy with increasing value assignment. We can observe that the solver we are proposing is, on average, comparable with the state-of-the-art.

Comparing GPU vs CPU. In this section we compare the GPU parallel version of the solver w.r.t. a purely sequential version. The core of the propagators are implemented in the same way (i.e., they use the same C encoding). The main drawbacks of the GPU computations are primarily related to data transfers, due to the GPU memory latency and coalesced access patterns, and to the difference between the GPU clock and the CPU clock.

We have tested various benchmarks described in Table 1: the running times are comparable for the sequential and parallel executions. Similar considerations hold for other "small" instances. We used the upper bound (UB) parameter

N	CPU	GPU	Gecode	JaCoP
24	6.273	9.699	7.094	47.59
26	5.975	8.773	7.438	47.55
28	50.88	68.47	66.88	442.6
30	930.3	1278	1407	9600

N	B	CPU	GPU	Gecode	JaCoP
40	4	88.59	84.75	19.02	2.570
41	4	92.92	90.71	19.54	2.610
42	4	97.03	95.41	20.54	2.700
43	4	108.4	98.75	21.35	2.850

k	n	m	CPU	GPU	Gecode	JaCoP
10	20	200	0.043	0.053	0.696	2.550
10	20	300	0.068	0.082	1.740	4.730
10	20	400	0.175	0.159	3.155	8.460
10	20	500	0.339	0.306	4.968	13.94

Table 1: Comparison between **i-solv** (sequential CPU and parallel CPU versions), Gecode, and JaCoP for the *nQueens*, *Schur*, and *propagation stress* benchmarks.

to move constraints from the host queue to the device queue. UB is calculated empirically, and it is automatically set by the solver in a preprocessing step, by considering the average numbers of global memory accesses w.r.t. the type of propagators involved in the model. For example, if there is an average of 3 memory accesses for each propagator, and each propagator requires $\mathcal{O}(1)$ time, then the upper bound will be set to at least 900, since each global memory access requires about 300 clock cycles. Table 2 shows how the UB affects the computational time on the *Golomb ruler* problem for a ruler of 20 integers. Notice that the solver with an appropriate upper bound performs better than both the CPU and the GPU without upper bound (UB = 0, all constraints propagated on device). The model comprise both $\mathcal{O}(1)$ and $\mathcal{O}(n)$ propagators.

CPU	UB = 0	UB = 100	UB = 500	UB = 1000	UB = 1500	UB = 2000
266.4	223.4	216.4	214.2	210.4	207.8	208.2

Table 2: Influence of the upper bound parameter on the *Golomb ruler* problem.

Significant performance improvements emerge when more complex constraints are considered. As explained in Section 4, the GPU is delegated to large sets of non trivial propagators. Using the CUDA framework, the CPU and the GPU can execute concurrently, since the kernels and the memory copy operations between host and device can be performed asynchronously. Let us focus on two "expensive" constraints, namely the *inverse* and the combinatorial *table* constraint.

The *inverse* constraint. This constraint ties two arrays of variables using the global *inverse* property. Given two lists $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_n]$ of integer variables, where $D^{x_i} = D^{y_i} = [1..n]$, the constraint $inverse(X, Y)$ holds iff $(\forall i \in [1..n])(\forall j \in [1..n])(x_i = j \leftrightarrow y_j = i)$. The FlatZinc implementation of this constraint uses n^2 Boolean variables and $2n^2$ *reified equality* constraints:

$$\bigwedge_{i,j} x_i = j \leftrightarrow B_{ij} \wedge \bigwedge_{i,j} y_j = i \leftrightarrow B_{ij}$$

The GPU version of this constraint is implemented by $2n$ propagators. Namely, n propagators are used for the "$\rightarrow$" (resp., "$\leftarrow$") direction of the constraint, considering the labeling of one variable in X (resp., in Y). Since we expand the relation $x_i = j \leftrightarrow y_j = i$ either on the left or the right side depending on

the labeled variable, we do not need to explicitly use the Boolean variables B_{ij} to link the binary equality constraints. These constraints are propagated by n threads. For example, let us assume that $x_1 = 2$ after the labeling of x_1; the constraint engine invokes the propagator $inverse(x_1, Y)$ where the thread whose $threadIdx = 2$ propagates the constraint $y_2 = 1$ (i.e, $B_{12} = \mathbf{true}$), while the other threads propagate the constraints $y_i \neq 1$, where $i \in \{1, 3, 4, \ldots, n-1, n\}$.

Table 3 compares the sequential and the parallel implementations of the *inverse* constraints, by increasing the number n of variables in its scope.

n	CPU	GPU	Speedup
100	0.030	0.026	1.15
250	0.338	0.152	2.22
500	2.456	0.744	3.30
750	7.855	2.142	3.66

Table 3: Time comparison for the *inverse* constraint.

For $n = 100$ there is a poor speedup, since the CPU cores are faster than the GPU cores and the instance of the problem is small. The speedup increases for bigger instances (i.e., $n > 200$) where the parallel computations offset the difference of speed between CPU and GPU cores. We have verified that the FlatZinc encoding of the *inverse* constraint is sensibly slower; for instance, if $n = 100$, the CPU takes time 3.583 seconds, while the GPU 3.334 seconds.

The *inverse* constraint is employed in several encodings, such as the *black hole* problem, and it is also used to create the dual models of problems.

The *table* constraint. A *table constraint* is an *extensional* constraint defined by explicitly listing (a set of n) m-tuples of values that are either allowed (*positive* table constraint) of disallowed (*negative* table constraint) for the variables in its scope. *Table* constraints arise naturally in configuration problems where they represent available combinations of options. For some applications, compatibility between resources, e.g., persons or machines, can be expressed by tables. Tabular data may also come from databases: the results of database queries are sometimes expressed as tables that have large arity.

A table constraint c represented by a $n \times m$ matrix and the *Generalized Arc Consistency* (GAC) [19] is maintained through propagation. Precisely, focusing on a variable $x_i \in \{x_1, \ldots, x_m\} = \text{scp}(c)$ a *support* for all the values in D^{x_i} is searched. This is realized by iterating over the n allowed tuples until a valid one is found. This algorithm ensures consistency in time $\mathcal{O}(n^m)$ (a faster, but more complex, algorithm is presented in [12]).

Using the GPU, it is possible to reduce this time to (parallel) time $\mathcal{O}(1)$, by performing the GAC test as follows: we assign each row to a kernel block, and each column to a different thread within the block. For table constraints with scope size larger than 1024, we split the computation among multiple kernels. For $1 \leq i \leq n$ and $1 \leq j \leq m$, thread t_{ij} checks whether the value contained in the cell c_{ij} is valid w.r.t. the domain D_{x_j}. The domains of the variables involved in the constraint are then replaced with the (new) domains, containing only those values that still might lead to a solution, as determined by each block.

We impose a specific ordering among propagated constraints: we first propagate binary constraints and constraints that have a fast propagator, that may eventually lead to a failure; more expensive propagators are executed last.

Table 4 compares the times for the propagation of the *table* constraint varying the number of rows n, the number of columns m, and the size of the domains of the variables. The tables are filled with random values, where $|D|$ is the size of the domain; note that larger domains produce fewer valid tuples after the labeling of a variable involved in the constraint.

$n \times m$	$\mid D \mid$	CPU	GPU	Speedup	$n \times m$	$\mid D \mid$	CPU	GPU	Speedup
100×100	2	0.002	0.001	2.00	100×100	50	0.001	0.001	1.00
250×250	2	0.007	0.003	2.30	250×250	50	0.003	0.001	3.00
500×500	2	0.026	0.010	2.60	500×500	50	0.013	0.004	3.25

Table 4: Time comparison for the *table* constraint with random values.

Examples containing table and inverse constraints.
The *Three-barrels* problem is a planning problem, where the state of the world is represented by three barrels of wine, whose capacities are n (even number), $n/2+1$, and $n/2-1$, respectively. At the beginning, the largest barrel is full of wine, while the other two are empty. The goal is to reach a state in which the two largest barrels contain the same amount of wine. Moreover, the only admissible action is to pour wine from one barrel to another, until the latter is full or the former is empty. We encoded this problem as a decision problem, by imposing an upper bound ℓ on the number of actions and evaluating whether the goal state can be reached in ℓ steps. In this setting, we have $3(\ell+1)$ variables, with domains $\{1,\ldots,n\}$, representing the sequence of states, and ℓ variables with domains $\{0,\ldots,5\}$, representing the 6 possible "pouring" actions. The labeling is done on the action variables, and ℓ table constraints tie the i^{th} state with the successor $i+1^{th}$ state. Table 6 (left) shows the results for the *Three-barrels* problem considering a number of actions ℓ equal to n, that was experimentally found to be the length of the shortest successful plan. The speedup is slowly increasing due to the size of the tables ($r \times 7$, with r proportional to n) and the number of valid rows at each labeling (at most 6 given the current state), that reduce the propagation time to $\mathcal{O}(r)$.
The *Black-hole* is a card game problem derived from [4]. A MiniZinc model is also present in the benchmark folder of the MiniZinc distribution [15], using both the global constraints *inverse* and *table*. The former is used to relate card values and positions in the sequence, while the latter is used to impose matching constraints among consecutive cards. The $<$ constraints impose an order between played cards, and are always propagated on the host. Table 6 (right) shows the results for the *Black-hole* game problem. Since the game is devised for 52 cards, the set of order constraints for instances 104 and 208 are artificially introduced. The table shows an increasing speedup. The GPU is faster even on small instances, since the two expensive constraints are propagated in parallel on the GPU.

Three-Barrels Problem			
n	CPU	GPU	Speedup
100	176.5	160.8	1.09
120	364.9	324.3	1.12
140	679.6	588.8	1.15

Black-hole Problem			
n. cards	CPU	GPU	Speedup
52	7.637	7.694	0.99
104	68.14	51.08	1.33
208	73.77	42.66	1.72

Table 5: Time comparison for the *Three-barrels* problem and the *Black-hole* game

Positive table *constraint benchmarks.* The following benchmark problems are defined using only positive table constraints.[5] They include some well-known problems, such as the *crossword* game, the *Langford* problem, several synthetic problems, and some other real-world problems, such as the *modified Renault* problem. A speedup of at least 2 is obtained in all the problem instances, showing that the use of the GPU pays off on large instances and real problems.

Instance	CPU	GPU	Speedup	Instance	CPU	GPU	Speedup
CW-m1c-lex-vg4-6	0.015	0.005	3.00	langford-2-50	44.06	15.16	2.94
CW-m1c-uk-vg16-20	1.488	0.225	6.61	ModRen_0	0.381	0.154	2.74
CW-m1c-lex-vg7-7	209.4	43.87	4.77	ModRen_49	0.317	0.117	2.74
langford-2-40	136.4	46.39	2.90	RD_k5_n10_d10_m15	0.138	0.053	2.60

Table 6: Positive *table* constraint benchmarks.

6 Related Work

Extensive research has been conducted focusing on parallelizing backtracking search, both in the context of CSP as well as in more general search-based scenarios (e.g., [28, 9, 11, 25]). Some works in this direction include the foundational work of Van Hentenryck in parallelizing the Chip system [26], the follow-up work in various CLP systems (e.g., [6]), the work of Perron [17], Schulte [22], and the more recent explorations by Michel et al. [14].

The problem of parallelizing consistency techniques has been also explored in the literature. The seminal works of Nguyen and Deville [16] and Hamadi [7] present methods based on message passing and distributed memory platforms; these approaches rely on the partitioning of the set of constraints among processors, and the use of messages to exchange variable domains. More recent approaches shifted the focus to multicore platforms and multithreaded implementations — e.g., the proposals by Rolf and Kuchcinksi [18] and Ruiz-Andino et al. (focused on non-binary constraints [20]). Note that, following the results from Kasif [8], establishing arc-consistency is P-complete; this is an indication that extracting parallelism from AC is, in general, not an easy problem (and, in the worst case, may not lead to complexity improvements).

To the best of our knowledge, this is the first reported effort exploring the use of GPGPUs in constraint propagation; some related effort includes [2], that

[5] These benchmarks can be downloaded from `http://becool.info.ucl.ac.be/resources/positive-table-constraints-benchmarks`.

shows how to parallelize unit propagation on GPGPUs. Some preliminary studies have instead addressed the problem of parallelizing *search* on GPUs [2, 5, 13].

7 Future work and Conclusions

In this paper, we presented a feasibility study exploring the potential for exploitation of fine-grained GPU-level parallelism from the process of constraint propagation. The investigation has been grounded in a prototype (with competitive performance with the state-of-the-art), demonstrating the potential for enhanced performance, especially in the context of complex global constraints. This is not an easy task, and the speedups proposed are in-line with results observed for parallelization of other classes of problems on GPUs.

This work complements preliminary studies [2, 27], conducted by the authors, in the context of SAT and ASP solving—where we demonstrated performance improvement from the "orthogonal" direction of parallelizing the actual search process. The combination of these two aspects (parallel search and parallel propagation) provide a roadmap for the creation of a fully GPU-parallel constraint solver—which is the focus of our future effort. The performance improvements for complex constraints reflect also on the potential for effective exploitation of parallelism in the case of domain-specific constraints with complex propagation strategies. We experimented with an ad-hoc constraint-based implementation of protein structure prediction via fragment assembly, parallelized on GPUs using similar techniques, with excellent performance results, outperforming previous approaches [1, 3]. We will continue along our current efforts of developing ad-hoc strategy to propagate complex constraints on GPUs.

Let us conclude with a final observation: the overall strategy for handling constraint propagation reported in Algorithm 2 is designed for efficient sequential implementation, and indeed is at the core of the state-of-the-art constraint solvers. Alternative schemes (e.g., AC-3), that can be found in several other implementations, provide a lower level of sequential performance, but they are also more amenable for GPU-level parallelization (as we demonstrated in a preliminary study). Unfortunately, the difference in sequential performance effectively defeats the advantages gained from parallelism.

Acknowledgments. The authors acknowledge Marco Meneghin for his support in the developing of the wrapper from FlatZinc.

References

1. F. Campeotto, A. Dovier, and E. Pontelli. Protein structure prediction on GPU: a declarative approach in a multi-agent framework. In *Proc. of International Conference on Parallel Processing*. IEEE, pp. 474–479, 2013.
2. A. Dal Palú, A. Dovier, A. Formisano, and E. Pontelli. Exploiting unexploited computing resources for computational logics. In *9th Italian Convention on Computational Logic*, CEUR Workshop Proceedings, Vol. 857, pp. 74–88, 2012.

3. A. Dal Palú, A. Dovier, F. Fogolari, and E. Pontelli. CLP-based protein fragment assembly. TPLP, 10(4–6):709–724, 2010.
4. I. Gent et al. Search in the Patience Game 'Black Hole'. *AI Communications*, 20(3):211–226, 2007.
5. K. Gulati and S.P. Khatri. Boolean Satisfiability on a Graphic Processor. In *Great Lakes Symposium on VLSI*, pp. 123–126. ACM, 2010.
6. G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.
7. Y. Hamadi. Optimal Distributed Arc Consistency. *Constraints*, 7(3-4), 2002.
8. S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, 1990.
9. H. Kitano and J.A. Hendler, editors. *Massive Parallel Artificial Intelligence*. AAAI/MIT Press, 1994.
10. K. Kuchcinski and R. Szymanek. JaCoP Library User's Guide, 2012. `http://jacop.osolpro.com/`.
11. H. Le and E. Pontelli. Dynamic Scheduling in Parallel Answer Set Programming Solvers. In *High Performance Computing Symposium*. ACM Press, 2007.
12. C. Lecoutre. STR2 Optimized Simple Tabular Reduction for Table Constraints. *Constraints*, 16(1), 2011.
13. Q. Meyer, F. Schonfeld, M. Stamminger, and R. Wanka. 3-SAT on CUDA: Towards a Massively Parallel SAT Solver. In *HPCS*, pp. 306–313. IEEE, 2010.
14. L. Michel, A. See, and P. Van Hentenryck. Transparent Parallelization of Constraint Programming. *INFORMS Journal on Computing*, 21(3), 2009.
15. N. Nethercote et al. MiniZinc: Towards a Standard CP Modelling Language. In *Proc. of CP 2007*, pp. 529–543. Springer, 2007. `www.minizinc.org`.
16. T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30(1–2):227–250, 1998.
17. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Principles and Practice of Constraint Programming*, pp. 346–360, Springer, 1999.
18. C. Rolf and K. Kuchcinski. Parallel Consistency in Constraint Programming. In *Proc. of PDPTA*, pp. 638–644, CSREA Press, 2009.
19. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier, 2006.
20. A. Ruiz-Andino, L. Araujo, F. Saenz, and J. Ruz. Parallel Execution Models for Constraint Propagation. In *Proc. of CP*, LNCS 1520, p. 473, Springer Verlag, 1998.
21. J. Sanders and E. Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.
22. C. Schulte. Parallel Search Made Simple. In *Techniques for Implementing Constraint Programming Systems*, TRA9/00, University of Singapore, 2000.
23. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *ACM TOPLAS*, 31(1), 2008.
24. C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and Programming with Gecode, 2013. Web site: `http://www.gecode.org`.
25. E.G. Talbi. *Parallel Combinatorial Optimization*. John Wiley and Sons, 2006.
26. P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *Proc. of ICLP*, pp. 165–180, MIT Press, 1989.
27. F. Vella, A. Dal Palù, A. Dovier, A. Formisano, and Enrico Pontelli. CUD@ASP: Experimenting with GPGPUs in ASP solving. In *10th Italian Convention on Computational Logic*, CEUR Workshop Proceedings, Vol. 1068, pp. 163–177, 2013.
28. H. Zhang, M.P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *JSC*, 21(4):543–560, 1996.