Parallel Execution of the ASP Computation - an Investigation on GPUs∗

Agostino Dovier[1], Andrea Formisano[2], Enrico Pontelli[3], Flavio Vella[4]
[1]*Dip. di Matematica e Informatica, Università di Udine*
[2]*Dip. di Matematica e Informatica, Università di Perugia*
[3]*Dept. of Computer Science, New Mexico State University*
[4]*IAC-CNR and Dip. di Informatica, Sapienza Università di Roma*

submitted 29 April 2015; accepted 5 June 2015

Abstract

This paper illustrates the design and implementation of a conflict-driven ASP solver that is capable of exploiting the *Single-Instruction Multiple-Thread* parallelism offered by *General Purpose Graphical Processing Units (GPUs)*. Modern GPUs are multi-core platforms, providing access to large number of cores at a very low cost, but at the price of a complex architecture with non-trivial synchronization and communication costs. The search strategy of the ASP solver follows the notion of *ASP computation*, that avoids the generation of unfounded sets. Conflict analysis and learning are also implemented to help the search. The CPU is used only to pre-process the program and to output the results. All the solving components, i.e., nogoods management, search strategy, (non-chronological) backjumping, heuristics, conflict analysis and learning, and unit propagation, are performed on the GPU by exploiting SIMT parallelism. The preliminary experimental results confirm the feasibility and scalability of the approach, and the potential to enhance performance of ASP solvers.

KEYWORDS: ASP solvers, ASP computation, SIMT parallelism, GPU computing

1 Introduction

Answer Set Programming (ASP) (Marek and Truszczynski 1998; Niemelä 1999) has gained momentum in the Logic Programming and Artificial Intelligence communities as a paradigm of choice for a variety of applications. In comparison to other non-monotonic logics and knowledge representation frameworks, ASP is syntactically simpler and, at the same time, very expressive. The mathematical foundations of ASP have been extensively studied; in addition, there exist a large number of building block results about specifying and programming using ASP (see (Gelfond 2007) and the references therein). ASP has offered novel and highly declarative solutions in a wide variety of application areas, including planning, verification, systems diagnosis, semantic web services composition and monitoring, and phylogenetic inference. An important push towards the popularity of ASP has come from the development of very efficient ASP solvers, from SMODELS (Syrjänen and

∗ Research partially supported by INdAM GNCS-14, GNCS-15 projects and NSF grants DBI-1458595, HRD-1345232, and DGE-0947465. Hardware partially supported by NVIDIA. We thank Massimiliano Fatica for the access to the Titan cards and Alessandro Dal Palù for the useful discussions.

Niemelä 2001) to CLASP (Gebser et al. 2012a), among many others. In particular, systems like CLASP and its variants have been shown to be competitive with the state-of-the-art in several domains, including competitive performance in SAT solving competitions. In spite of the efforts in developing fast execution models for ASP, execution of large programs and programs requiring complex search patterns remains a challenging task, limiting the scope of applicability of ASP in certain domains (e.g., planning).

In this work, we explore the use of parallelism as a viable approach to enhance performance of ASP inference engines. In particular, we are interested in devising techniques that can take advantage of recent architectural developments in the field of *General Purpose Graphical Processing Units (GPUs).* Modern GPUs are multi-core platforms, offering massive levels of parallelism; vendors like AMD and NVIDIA support the use of GPUs for general-purpose non-graphical applications, providing dedicated APIs and development environments. Languages and language extensions like *OpenCL* (Khronos Group Inc 2015) and *CUDA* (NVIDIA Corporation 2015) support the development of general purpose applications on GPUs. To the best of our knowledge, the use of GPUs for ASP computations has not been explored and, as demonstrated in this paper, it opens an interesting set of possibilities and issues to be resolved. It is a contribution of this paper to bring these opportunities and challenges to the attention of the logic programming community.

The work proposed in this paper builds on two existing lines of research. The exploitation of parallelism from ASP computations has been explored in several proposals, starting with the seminal work presented in (Pontelli and El-Khatib 2001; Finkel et al. 2001), and later continued in several other projects (e.g., (Balduccini et al. 2005; Pontelli et al. 2010; Gebser et al. 2012b; Perri et al. 2013)). Most of the existing proposals have primarily focused on parallelization of the search process underlying the construction of answer sets, by distributing parts of the search tree among different processors/cores (*search parallelism*). Furthermore, the literature has focused on parallelization on traditional multi-core or Beowulf architectures. These approaches are not applicable in the context of GPUs since the models of parallelization used on GPUs are deeply different, as we will discuss in the paper. GPUs are designed to operate with very large number of very lightweight threads, operating in a synchronous way; GPUs present a significantly more complex memory organization, that has great impact on parallel performance, and existing parallel ASP models are not directly scalable on GPUs. The second line of research that supports the effort proposed in this paper can be found in the recent developments in the area of GPUs for SAT solving and constraint programming. The work in (Dal Palù et al. 2012; Dal Palù et al. 2015) illustrates how to parallelize the search process employed by the DPLL procedure in solving a SAT problem on GPUs; the outcomes demonstrate the potential benefit of delegating to GPUs tails of the branches of the search tree—an idea that we have also investigated in a previous version of the present work (Vella et al. 2013). Several other proposals have appeared in the literature suggesting the use of GPUs to parallelize parts of the SAT solving process—e.g., the computation of variable heuristics (Manolios and Zhang 2006). In the context of constraint programming, (Campeotto et al. 2014, PADL) explores the parallelization on GPUs of constraint propagation, in particular in the case of complex global constraints; (Campeotto et al. 2014, ECAI) shows how (approximated) constraint-based local-search exploits the parallelism of the GPU to concurrently visit larger neighborhoods, improving

the quality of the results. In (Campeotto et al. 2015), a GPU-based constraint solver is used for fast prediction of protein structures.

The lesson learned from the above studies, and explored in this paper, is that, as long as (exact) search problems are concerned, the best way to exploit GPUs for search problems is the parallelization of the "easy" activities, such as constraint propagation. Therefore, the main focus of this work is to exploit GPU parallelism for addressing the various (polynomial time) operations associated to answer set search, such as unit propagation, heuristics, conflict analysis, management of nogoods, and learning. The control of the search is also handled by the GPU, but the visit of the search tree is not parallelized. (As we will see, the CPU is used only to read and pre-process the program and to output the results.) This approach contrasts sharply with that of (Dal Palù et al. 2015), which achieves the parallelization of a DPLL procedure mainly through search space partitioning and distribution of entire search subtrees to different single threads.

The usual algorithms for unfounded set check (Gebser et al. 2012a), which is part of the CLASP implementation, are hard to be efficiently implemented exploiting fine-grained SIMT parallelism, also due to the *reachability bottleneck* (Khuller and Vishkin 1994). Approaches relying on a coarse-grained parallelism, such as, for instance, running several instances of the source pointers algorithm (Simons et al. 2002) to process separately each strongly connected component of the dependency graph, are in a sense orthogonal to our approach. We have circumvented this problem by implementing the search strategy described in (Liu et al. 2010), that was also successfully used for solving ASP programs delaying the grounding process (Dal Palù et al. 2009). In spite of some of these limitations, the current results already show the scalability and feasibility of the overall approach.

2 Background

2.1 Answer Set Programming

In this section, we will review the basic notions on ASP (e.g., (Marek and Truszczynski 1998; Niemelä 1999)). We focus on the classical single-head clauses. Let us consider a language composed of a set of propositional atoms $\mathcal{P}$. An ASP rule has the form

$$p_0 \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n \tag{1}$$

where $n \geq 0$ and $p_i \in \mathcal{P}$. A rule that includes first-order atoms with variables is simply seen as a syntactic sugar for all its ground instances. p_0 is referred to as the *head* of the rule ($head(r)$), while the set of atoms $\{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ is referred to as the body of the rule ($body(r)$). In particular, $body^+(r) = \{p_1, \ldots, p_m\}$ and $body^-(r) = \{p_{m+1}, \ldots, p_n\}$. A *constraint* is a rule of the form:

$$\leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n \tag{2}$$

A *fact* is a rule of the form (1) with $n = 0$. A program Π is a collection of ASP rules. We will use the following notation: $atom(\Pi)$ denotes the set of all atoms in Π, while $rules(p) = \{r : head(r) = p\}$ denotes the set of all the rules that define the atom p.

Let Π be a program; its *positive dependence graph* $\mathcal{D}^+_\Pi = (V, E)$ is a directed graph with $V = atom(\Pi)$ and the edges are $E = \{(p, q) : r \in \Pi, head(r) = p, q \in body^+(r)\}$.

In particular, we are interested in recognizing cycles in $\mathcal{D}^+_\Pi$; the number of non-self loops in $\mathcal{D}^+_\Pi$ is denoted by $loop(\Pi)$. A program Π is *tight* (resp., *non-tight*) if $loop(\Pi) = 0$ (resp., $loop(\Pi) > 0$). A *strongly connected component* of $\mathcal{D}^+_\Pi$ is a maximal subgraph X of $\mathcal{D}^+_\Pi$ such that there exists a directed path between each pair of nodes in X.

The *semantics* of ASP programs is provided in terms of *answer sets*. An *interpretation* is a set M of atoms. M is an answer set of a program Π if it is the subset-minimal model of the *reduct program* Π^M (we refer the reader to (Baral 2010) for a detailed treatment).

2.2 Answer Set Computation

Let us recall some of the techniques that we will exploit in our proposal, namely the notion of ASP computation (Liu et al. 2010) and some of the techniques used by CLASP (Gebser et al. 2012a). Let us start with the latter. The CLASP system explores a search space composed of all truth value assignments to the atoms in Π, organized as a binary tree. The successful construction of a branch in the tree corresponds to the identification of an answer set of the program. If a (possibly partial) assignment fails to satisfy the rules in the program, then backjumping procedures are used to backtrack to the node in the tree that caused the failure. The design of the tree construction and the backjumping procedure in CLASP are implemented in such a way to guarantee that if a branch is successfully constructed, then the outcome will be an answer set of the program. CLASP's search is also guided by special assignments of truth values to subsets of atoms that are known not to be extendable into an answer set—these are referred to as *nogoods* (Dechter 2003; Rossi et al. 2006). Assignments and nogoods are sets of assigned atoms—i.e., entities of the form Tp or Fp, denoting that p has been assigned `true` or `false`, respectively. For assignments it is also required that, for each atom p, at most one between Tp and Fp is present. Given an assignment A, let $A^T = \{p : Tp \in A\}$ and $A^F = \{p : Fp \in A\}$. Note that A^T is an interpretation. A *total* assignment A is such that, for every atom p, $\{Tp, Fp\} \cap A \neq \emptyset$. Given a (possibly partial) assignment A and a nogood δ, we say that δ is *violated* if $\delta \subseteq A$. In turn, A is a *solution* for a set of nogoods Δ if no $\delta \in \Delta$ is violated by A. The concept of nogood can be used during deterministic propagation phases (*unit propagation*) to determine additional assignments. Given a nogood δ and a partial assignment A such that $\delta \setminus A = \{Fp\}$ (resp., $\delta \setminus A = \{Tp\}$), then we can infer the need to add Tp (resp., Fp) to A in order to avoid violation of δ.

We distinguish two types of nogoods. The *completion nogoods* (Fages 1994) are derived from Clark completion of a logic program; we will denote with $\Delta_{\Pi_{cc}}$ the set of completion nogoods for the program Π. The *loop nogoods* Λ_Π (Lin and Zhao 2004) are derived from the loop formulae of Π. Let Π be a program and A an assignment (Gebser et al. 2012a):

- If Π is tight, then $atom(\Pi) \cap A^T$ is an answer set of Π iff A is a solution of $\Delta_{\Pi_{cc}}$.
- If Π is not tight, then $atom(\Pi) \cap A^T$ is an answer set of Π iff A is a solution of $\Delta_{\Pi_{cc}} \cup \Lambda_\Pi$.

In this paper we focus on the completion nogoods (although the solver might deal with the other as well). Let us define the Clark completion Π_{cc} of a program Π. For each rule $r \in \Pi$: $head(r) \leftarrow body(r)$ we add to Π_{cc} the formulae

$$\beta_r \quad \leftrightarrow \quad \tau_r, \eta_r \qquad \tau_r \quad \leftrightarrow \quad \bigwedge_{a \in body^+(r)} a \qquad \eta_r \quad \leftrightarrow \quad \bigwedge_{b \in body^-(r)} \neg b \quad (3)$$

where β_r, τ_r, η_r are new atoms. For each $p \in atom(\Pi)$, the following formula is added to Π_{cc} (if $rules(p) = \emptyset$, then the formula reduces simply to $\neg p$):

$$p \quad \leftrightarrow \quad \bigvee_{r \in rules(p)} \beta_r \tag{4}$$

The *completion nogoods* reflect the structure of the implications in the formulae in Π_{cc}:

- From the first formula above we have the nogoods: $\{F\beta_r, T\tau_r, T\eta_r\}, \{T\beta_r, F\tau_r\}$, and $\{T\beta_r, F\eta_r\}$.
- From the second and third formula above we have the nogoods: $\{T\tau_r, Fa\}$ for each $a \in body^+(r)$; $\{T\eta_r, Tb\}$ for each $b \in body^-(r)$; $\{F\tau_r\} \cup \{Ta : a \in body^+(r)\}$; and $\{F\eta_r\} \cup \{Fa : b \in body^+(r)\}$.
- From the last formula we have the nogoods: $\{Fp, T\beta_r\}$ for each $r \in rules(p)$ and $\{Tp\} \cup \{F\beta_r : r \in rules(p)\}$.

$\Delta_{\Pi_{cc}}$ is the set of all the nogoods defined as above plus the constraints (2) that introduce nogoods of the form $\{Tp_1, \ldots, Tp_m, Fp_{m+1}, \ldots, Fp_n\}$.

The work described in (Liu et al. 2010) provides a *computation-based* characterization of answer sets for programs with abstract constraints. One of the outcomes of that research is the development of a computation-based view of answer sets for logic programs; we will refer to this model as ASP COMPUTATIONS. The computation-based characterization is based on an incremental construction process, where the choices are performed at the level of what rules are actually applied to extend the partial answer set. Let T_Π be the immediate consequence operator of Π: if I is an interpretation, then

$$T_\Pi(I) \quad = \quad \left\{ \begin{array}{lll} p_0 & : & p_0 \leftarrow p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n \in \Pi \wedge \\ & & \{p_1, \ldots, p_m\} \subseteq I \wedge \{p_{m+1}, \ldots, p_n\} \cap I = \emptyset \end{array} \right\} \tag{5}$$

An *ASP Computation* of a program Π is a sequence of interpretations $I_0 = \emptyset, I_1, I_2, \ldots$ satisfying the following conditions:

- $I_i \subseteq I_{i+1}$ for all $i \geq 0$ (*Persistence of Beliefs*)
- $I_\infty = \bigcup_{i=0}^{\infty} I_i$ is such that $T_\Pi(I_\infty) = I_\infty$ (*Convergence*)
- $I_{i+1} \subseteq T_\Pi(I_i)$ for all $i \geq 0$ (*Revision*)
- if $a \in I_{i+1} \setminus I_i$ then there is a rule $a \leftarrow body$ in Π such that $I_j \models body$ for each $j \geq i$ (*Persistence of Reason*).

I_0 can be the empty set or, more in general, a set of atoms that are logical consequences of Π. We say that a computation $I_0, I_1, \ldots$ converges to I if $I = \bigcup_{i=0}^{\infty} I_i$. Liu et al. (2010) prove that given a ground program Π, an interpretation I is an answer set of Π if and only if there exists an ASP computation that converges to I. I determines the assignment A such that $A^T = I$ and $A^- = atom(\Pi) \setminus I$.

2.3 CUDA

GPU computing is a general term indicating the use of the multicores available within modern graphical processing units for general purpose parallel computing. NVIDIA is one of the pioneering manufacturers in promoting GPU computing, especially thanks to its *Computing Unified Device Architecture (CUDA)* (NVIDIA Corporation 2015). A GPU is composed of a collection of *Streaming MultiProcessors (SMs);* in turn, each SM contains a

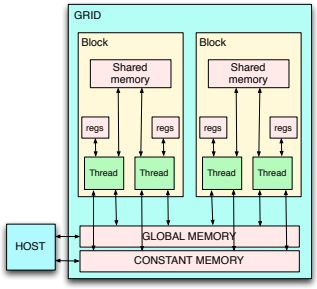

Fig. 1. CUDA Logical Architecture

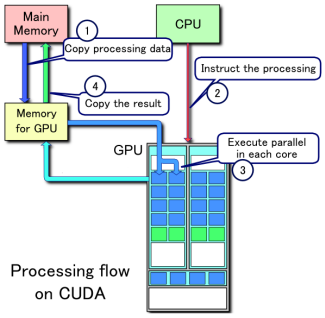

Fig. 2. Generic workflow in CUDA

collection of computing cores (e.g., 32 in the Fermi platforms). Each GPU provides access to both on-chip memory (used for thread registers and shared memory) and off-chip memory (used for L2 cache, global memory and constant memory). The architecture of the GPU also determines the *GPU Clock* and the *Memory Clock* rates. The underlying conceptual 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, while data and operands may differ from thread to thread. CUDA's architectural model is represented in Fig. 1. 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*). The host program contains all the instructions necessary to initialize the data in the GPU, define the number of threads and manage the kernels. A kernel is a set of instructions to be executed by many concurrent threads running on the GPU. The programmer organizes these threads in thread *blocks* and *grids* of blocks. Each thread in a block executes an instance of the kernel, and has a thread ID within its block. A grid is an array of blocks that execute the same kernel, read data input from the global memory, and write results to the global memory. When a CUDA program on the host launches a kernel, the blocks of the grid are distributed to the SMs with available execution capacity. The threads in the same block can share data, using shared high-throughput on-chip memory. The threads belonging to different blocks can only share data through the global memory. Thus, the block size allows the programmer to define the granularity of threads cooperation. Fig. 1 shows the CUDA threads hierarchy (Nickolls and Dally 2010). Referring to Fig. 2, a typical CUDA application can be summarized as follow:

Memory allocation and data transfer. Before being processed by kernels, the data must be copied to GPU's global memory. The CUDA API supports memory allocation (function `cudaMalloc()`) and data transfer to/from the host (function `cudaMemcpy()`).

Kernels definition. Kernels are defined as standard C functions; the annotation used to communicate to the CUDA compiler that a function should be treated as kernel has the form: `__global__ void kernelName (Formal Arguments).`

Kernels execution. A kernel can be launched from the host program using:

```
kernelName <<< GridDim, TPB >>> (Actual Arguments)
```

Algorithm 2.2 `CUD@ASP-computation`

Require: A set of nogoods Δ computed from the ground ASP program Π

1: $current\_dl := 1$ ▷ Initial decision level
2: $A := \emptyset$ ▷ Initial assignment is empty
3: $(A, \texttt{Violation}) :=$ **InitialPropagation**(A, Δ) ▷ CUDA kernel
4: **if** (`Violation` is true) **then return** no answer set
5: **else**
6: **loop**
7: $(\Delta_A, \texttt{Violation}) :=$ **NoGoodCheckAndPropagate**(A, Δ) ▷ CUDA kernels
8: $A := A \cup \Delta_A$;
9: **if** (`Violation` is true) $\wedge$ $(current\_dl = 1)$ **then return** no answer set
10: **else if** (`Violation` is true) **then**
11: $(current\_dl, \delta) =$ **ConflictAnalysis**(Δ, A) ▷ CUDA kernels
12: $\Delta := \Delta \cup \{\delta\}$
13: $A := A \setminus \{\overline{p} \in A \mid current\_dl < dl(\overline{p})\}$
14: **end if**
15: **if** (A is not total) **then**
16: $(\overline{p}, OneSel) :=$ **Selection**(Δ, A) ▷ CUDA kernel
17: **if** ($OneSel$ is true) **then**
18: $current\_dl := current\_dl + 1$
19: $dl(\overline{p}) := current\_dl$
20: $A := A \cup \{\overline{p}\}$
21: **else** $A := A \cup \{Fp : p \text{ is unassigned}\}$
22: **end if**
23: **else return** $A^T \cap atom(\Pi)$
24: **end if**
25: **end loop**
26: **end if**

where `GridDim` is the number of blocks of the grid and `TPB` specifies the number of threads in each block.

Data retrieval. After the execution of the kernel, the host needs to retrieve the results This is performed with another transfer operation from global memory to host memory.

3 Design of a conflict-based CUDA ASP Solver

In this section, we present the ASP solving procedure which exploits ASP computation, nogood handling, and GPU parallelism. The ground program Π, as produced by the grounder GRINGO, is read by the CPU. The current implementation accepts as inputs normal programs possibly extended with choice rules and cardinality rules. Choice and cardinality rules are eliminated in advance by applying the technique described in (Gebser et al. 2012). In this first prototype, we did not include weight rules and aggregates. The CPU computes the dependency graph of Π and its strongly connected components (using the classical Tarjan's algorithm), detecting, in particular, if the input program is tight. The CPU also computes the completion nogoods $\Delta_{\Pi_{cc}}$ and transfers them to the GPU. The CPU launches the various kernels summarized in Algorithm 2.2. The rest of the computation is performed completely on the GPU, under the control of the CPU. During this process, there are no memory transfers between the CPU and the GPU, with the exception of **(1)** flow control flags, such as the "exit" flag, used to communicate whether the computation is terminated, and **(2)** the transfer of the computed answer set from the GPU to the CPU.

The overall structure of Algorithm 2.2 is a conventional structure for an ASP solver. The differences lay in the selection heuristic (ASP computation) and in the parallelization of all

Main and auxiliary Kernels	blocks	threads
InitialPropagation	⌈ Num_Unitary_Nogoods/TPB ⌉	TPB
NoGoodCheckAndPropagate		
Binary Nogoods	Num_Rec_Assign_Vars	TPB*
Ternary Nogoods	Num_Rec_Assign_Vars	TPB*
General Nogoods	⌈ Num_Long_Nogoods/TPB ⌉	TPB
ConflictAnalysis		
MkNewNogood	1	1024
Backjump	⌈ Num_Tot_Atoms/TPB ⌉	TPB
Selection	1	TPB

Table 1. *Main CUDA kernels and the number of blocks-threads per kernel. TPB is the number of* threads-per-block *(heuristically set to 256). TPB* is the minimum between TPB and the number of nogoods immediately related to recently assigned variables—See Sect. 3.2.*

the support functions involved. Each atom of Π is identified with an index. The variable A represents the set of assigned atoms (with their truth values) computed so far. In practice, A is a vector such that: **(1)** $A[p] = 0$ iff the atom p is currently undefined; **(2)** $A[p] = i$, $i > 0$ (resp., $A[p] = -i$) means that atom p has been assigned true (resp., false) at the decision level i. The variable $current\_dl$ represents the current *decision level*; this variable acts as a counter that keeps track of the number of "choices" that have been made in the computation of an answer set. These variables are stored and updated in the GPU. A is transferred to the CPU as soon as an answer set is found. For each program atom p, the notation $\overline{p}$ represents the atom with a truth value assigned; $\neg\overline{p}$ denotes the complement truth value with respect to $\overline{p}$. The assignments in lines 8, 20, and 21 correspond to assignments of $A[p]$.

In the rest of the section, we focus on the various kernels executed by the procedure `CUD@ASP-computation` (Algorithm 2.2). Each kernel has its own number of blocks and of threads-per-block (TPB), as summarized in Table 1.

3.1 InitialPropagation

The set of nogoods Δ computed from the input program may include some unitary nogoods. A preliminary parallel computation partially initializes A by propagating them. Notice that the algorithm can be *restarted* several times—typically, this happens when more than one solution is requested. In such cases, the initial propagation also handles unit nogoods that have been learned in the previous executions. A single kernel is invoked with one thread for each unitary nogood. In particular, if k is the number of unitary nogoods, $\lceil k/TPB \rceil$ blocks, with TPB threads each, are started. The sign of the literal p in it is analyzed and the value array $A[p]$ is set consistently. If one thread finds $A[p]$ already assigned in a inconsistent way, the `Violation` flag is set to true and the computation ends.

The preliminary phase also includes a pre-processing step aimed at enabling a special treatment of binary and ternary nogoods, as described in the next section.

3.2 NoGoodCheckAndPropagate

Given a partial assignment A, each nogood δ needs to be analyzed to detect whether: **(1)** δ is violated by A or **(2)** δ is not violated but there is exactly one literal in it that is unassigned in A (i.e., $\delta \setminus A = \{\overline{p}\}$, where $\overline{p} = Fp$ or $\overline{p} = Tp$) then an inference must be executed, namely adding $\neg\overline{p}$ to A. The procedure is repeated until a fixpoint is reached.

To better exploit the SIMT parallelism and maximize the degree of thread concurrency, in each kernel execution the workload has to be divided among the threads as much uniformly as possible. To this aim, the set of nogoods is partitioned depending on their cardinality. Moreover, the **NoGoodCheckAndPropagate** is split in three steps, each one implemented by one different kernel. The first kernel deals with all the nogoods with exactly two literals, the second one processes the nogoods made of three literals and a third kernel processes all remaining nogoods. The first and the second kernels act as follows. The execution of each iteration of **NoGoodCheckAndPropagate** is driven by the atoms that have already been assigned a truth value; in particular, in the first iteration, the procedure relies on the atoms that have been assigned by the **InitialPropagation**. The motivation is that only the atoms with an assigned truth value may trigger either a conflict or a propagation of a nogood. New assignments contribute to the following steps, either by enabling further propagation or by causing conflicts. Thus, the first two kernels rely on a number of blocks that is equal to the number of assigned atoms. The threads in each block process the nogoods that share the same assigned atom. The number of threads of each block is established by considering the number of occurrences of each assigned atom in the binary (resp., ternary) nogoods. Observe that this number may change between two consecutive iterations of **NoGoodCheckAndPropagate**, and as such it is evaluated each time. Specific data structures (initialized once during the pre-processing phase) are used in order to determine, after each iteration of **NoGoodCheckAndPropagate** and for each assigned atom, which are the binary/ternary nogoods to be considered in the next iteration. We observe that, despite the overhead of performing such a pre-processing, this selective treatment of binary and ternary nogoods proved to be very effective in practice, leading to several orders of magnitude of performance improvement with respect to a "blind" approach that treats all nogoods in the same manner. The third kernel is called with one thread for each nogood of cardinality greater than three. If n is the number of such nogoods, the kernel runs $\lceil n/TPB \rceil$ blocks of TPB threads each. The processing of longer nogoods is realized by implementing a standard technique based on *watched literals* (Biere et al. 2009). In this kernel, each thread accesses the watched literals of a nogood and acts accordingly.

During unit propagation, atoms may be assigned either true or false truth values. Assignment of false is not a critical component in ensuring stability, while assignment of true might lead to unfounded solutions. The **Selection** procedure, defined in Sect. 3.4, introduces only positively assigned atoms of the form β_r. Each β_r occurs in nogoods involving the "head" p and the auxiliary variables τ_r and η_r. In the subsequent call to **NoGoodCheckAndPropagate**, Tp and $T\eta_r$ are introduced in A. They are positive literals but asserting them does not invalidate stability of the final model, if any. This is because their truth values are strictly related to the supportedness of p, due to the choice of β_r by the **Selection** procedure. Moreover, the assignments Fb for $b \in body^{-r}$ are added, but being negative, they do not lead to unfounded solutions.

There are two cases when a positive, unsupported, literal may be assigned by unit propagation: **(1)** when the propagating nogood comes from a constraint of the initial program, and **(2)** when the nogood has been learned. In these cases, the literal is propagated anyway, but it is marked as "unsupported". The ASP computation described earlier would suggest not to propagate such positive assignment. However, performing such a propagation helps in the early detection of conflicts and significantly speeds up the search for a solution. No-

tice that these inferences have to be satisfied in the computed solution, because they are consequences of the conjunction of all selected literals. When a complete assignment is produced, for each marked atom an inspection is performed to check whether support for it has been generated after its assignment. Thanks to specific information gathered during the computation this check is performed in constant time. If that is not the case, a restart is performed (for simplicity, this step is not depicted in Algorithm 2.2).

3.3 ConflictAnalysis Procedure

The **ConflictAnalysis** procedure is used to resolve a conflict detected by the **NoGoodCheckAndPropagate**; the procedure identifies a level dl and an assignment $\overline{p}$ the computation should backtrack to, in order to remove the nogood violation. This process allows classical backjumping in the search tree generated by Algorithm 2.2 (Russell and Norvig 2010; Rossi et al. 2006). This part of the solver is the one that is less suitable to SIMT parallelism, due to the fact that a (sequential) sequence of resolution steps must be encoded. This procedure ends with the identification of a unique implication point (UIP (Marques Silva and Sakallah 1999)) that determines the lower decision level/literal among those causing the detected conflicts. As default behavior, the solver selects one of the (possibly multiple) conflicts generated by **NoGoodCheckAndPropagate**. Heuristics can be applied to perform such a selection. In the current implementation priority is given to shorter nogoods. The kernel is run with a single block to facilitate synchronization—as we need to be sure that the first resolution step ends before the successive starts. The block contains a fixed number of threads (we use, by default, 1024 threads) and every thread takes care of one atom; if there are more atoms than threads involved in the learning, atoms are equally partitioned among threads. For each analyzed conflict, a new nogood is learned and added to Δ. This procedure takes also care of backtracking/backjumping, through the use of a specific kernel (**Backjumping**). The level in which to backjump is computed and the data structures (e.g., the values of A and of $current\_dl$) are updated accordingly. Notice that the prototype is capable of learning from all different conflicts detected by the same run of **NoGoodCheckAndPropagate**. The number of conflicts to process can be specified through a command-line option. In case of multiple learned nogoods involving different "target" decision levels, the lowest level is selected.

3.4 Selection Procedure

The purpose of this procedure is to determine an unassigned atom in the program. For each unassigned atom p occurring in the head of a clause in the original program, all nogoods reflecting the rule $\beta_r \leftarrow \tau_r, \eta_r$, such that $r \in rules(p)$ are analyzed to check whether $T\tau_r \in A$ and $F\eta_r \notin A$ (i.e., the rule is applicable). All p and all rules r that pass this test are evaluated according to a heuristic weight. Typical heuristics are used to rank the atoms. In particular, we consider the number of positive/negative occurrences of atoms in the program (by either simply counting the occurrences or by applying the *Jeroslow-Wang* heuristics) and the "activity" of atoms (Goldberg and Novikov 2007). Using a logarithmic parallel reduction scheme, the rule r with highest ranking is selected. Then, $T\beta_r$ is added to A. In the subsequent execution of **NoGoodCheckAndPropagate**, Tp and $F\eta_r$ are also

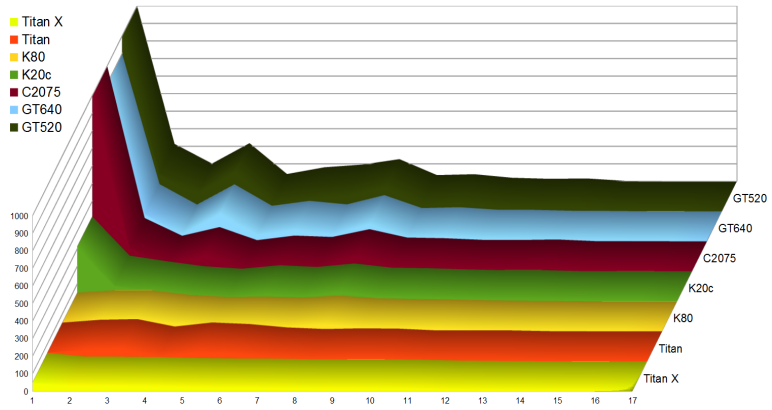

Fig. 3. A visualization of the experiments. The x axis reports the 17 instances (in decreasing order w.r.t. running time). The y axis shows the sum of the running times in seconds. The GPU models are shown on the z axis.

added to A. $F\eta_r$ imposes that all the atoms of $body^-(r)$ are set to false. This ensures the *persistence of beliefs* of the ASP computation.

4 Experimental Results and Conclusions

We have tested the software we have developed with seven NVIDIA GPUs with different computing capabilities. We report here just the number of cores and the GPU clock. Complete info can be retrieved from the vendor website.

GT520: GeForce: 48 cores, 1.62 GHz **GT640:** GeForce: 384 cores, 0.80 GHz
C2075: Tesla C2075: 448 cores, 1.15 GHz **K20c:** Tesla: 2496 cores, 0.71 GHz
K80: Tesla: 2496 cores, 0.82 GHz **Titan:** GeForce GTX TITAN (Fermi): 2688 cores, 0.88 GHz
Titan X: GeForce GTX TITAN X (Maxwell): 3072 cores, 1.08 GHz

We report here the results obtained for some of the benchmarks we used. Namely, those coming from the grounding of `stablemarriage-0-0`, `visitall-14-1`, `graph_colouring-125-0`, `labyrinth-11-0`, `ppm-70/90/120-0`, and `sokoban-15-1` programs and instances (material from ASP competitions (Alviano et al. 2013)). In Fig. 3 and Table 2 an excerpt the overall running times of 17 benchmarks is reported. As one might expect, the code scales with the card computing power. If $pow = \#cores \times GPUclock$ is chosen as an (approximated) index for the "power" of a card, a simple regression analysis on the sum of all running times shows us that the running time is $\approx -\frac{1}{2}pow + k$, where k is a constant depending on the overall number of instances tested. This is a good witness of scalability. Other analyses have been performed (using memory size and bitrate). Typically, memories with higher bitrate are installed in cards with larger numbers of cores. As a result, the bitrate "per single core" does not change significantly, and therefore this finer analysis did not lead us to different results.

Several aspects of the current implementation deserve further investigation. As mentioned earlier, the unfounded set check, which is part of the CLASP implementation, represents a challenge to parallel implementation, especially due to the *reachability bottleneck* (Kao and Klein 1993). A fast implementation of this module would allow us to

INSTANCE	GT 520	GT 640	C2075	K20c	K80	Titan	Titan X
0001-stablemarriage-0-0	20.90	10.15	9.65	11.82	12.46	5.87	10.20
0002-stablemarriage-0-0	42.77	32.90	7.16	16.18	26.65	51.69	20.47
0001-visitall-14-1	127.64	93.35	70.03	46.45	33.66	14.07	13.51
0003-visitall-14-1	218.64	155.28	82.85	30.36	38.02	28.74	25.27
0007-graph_colouring-125-0	214.12	155.44	133.88	90.64	63.54	66.49	28.89
0010-graph_colouring-125-0	17.82	4.65	11.56	7.89	4.12	6.25	3.39
0009-labyrinth-11-0	96.10	40.55	25.87	26.23	23.41	23.56	16.78
0023-labyrinth-11-0	–	899.34	–	313.60	50.57	50.67	49.54
0061-ppm-70-0	3.02	1.77	1.54	1.29	1.20	1.26	0.97
0072-ppm-70-0	4.11	3.27	2.96	3.06	2.60	2.62	2.06
0130-ppm-90-0	15.61	9.32	8.41	7.57	6.53	6.70	5.79
0153-ppm-90-0	2.25	1.65	1.46	1.33	1.10	1.24	0.94
0121-ppm-120-0	41.88	24.75	18.78	18.68	14.44	15.97	12.55
0128-ppm-120-0	0.96	0.79	0.70	0.76	0.63	0.68	0.50
0129-ppm-120-0	36.26	19.21	22.59	21.85	17.64	17.85	12.94
0167-sokoban-15-1	102.09	39.88	32.98	58.77	63.40	70.88	27.65
0589-sokoban-15-1	81.04	60.77	34.17	36.53	28.48	42.55	17.55

Table 2. *Performance of the GPU-based solver on instances from the 2014 ASP competition. Symbol '–' stands for 20 minutes timeout expiration.*

use safely other search heuristics combined with the ASP computation. Conflict-driven learning suffers from the same problem (and currently takes roughly 50% of the computation time). This is the real bottleneck, essentially because of the intrinsically serial algorithm used for conflict analysis. We are still working to speed-up this part as much as possible, but the current results already show the scalability and feasibility of the overall approach. Alternative approaches should be considered. A first attempt in developing alternative learning schemata, expressly conceived to benefit from SIMT parallelism, can be found in (Formisano and Vella 2014). Another possibility to be considered is the use of a GPU thread as the "host" of the main loop (that is currently assigned to the CPU). This would reduce the running time for the transfer of flags at the end of any kernel execution. On the other hand, *dynamic parallelism*—namely, the capability of calling a kernel from a kernel— is supported only by the most recent devices. Finally, another interesting approach to be considered is the execution of the final part of the search using an exhaustive search among all possible assignments for the last atoms (e.g., when 30-40 atoms are unassigned). This would exploit well the GPU parallelism. However, this would require the development of fast parallel algorithms for unfounded set extraction, in order to learn useful nogoods from non-stable total assignments.

To conclude, we have presented the first ASP solver running on GPUs and experimentally proved that the approach scales well on the power of the devices. The solver is not yet ready to challenge the best ASP solvers, e.g., CLASP (although it proves to be only 3 times slower on the sum of the running times of the 17 instances described above). Those solvers include families of heuristics for driving the search that are not (yet) implemented in our solver. A sensible speedup will be obtained as soon as the conflict analysis will be implemented by exploiting a truly parallel schema. Another point to be addressed is the development of an ad-hoc parallel handling of the cardinality constraints/aggregates by suitable kernels instead of removing them in the pre-processing stage, thus removing relevant structural information.

References

ALVIANO, M., CALIMERI, F., CHARWAT, G., DAO-TRAN, M., DODARO, C., IANNI, G., KRENNWALLNER, T., KRONEGGER, M., OETSCH, J., PFANDLER, A., PÜHRER, J., REDL, C., RICCA, F., SCHNEIDER, P., SCHWENGERER, M., SPENDIER, L. K., WALLNER, J. P., AND XIAO, G. 2013. The fourth answer set programming competition: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Proceedings*, P. Cabalar and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 8148. Springer, 42–53.

BALDUCCINI, M., PONTELLI., E., EL-KHATIB, O., AND LE, H. 2005. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing 31*, 6, 608–647.

BARAL, C. 2010. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

CAMPEOTTO, F., DAL PALÙ, A., DOVIER, A., FIORETTO, F., AND PONTELLI, E. 2014. Exploring the Use of GPUs in Constraint Solving. In *Proceedings of Practical Aspects of Declarative Languages - 16th International Symposium, PADL 2014*, M. Flatt and H. Guo, Eds. Lecture Notes in Computer Science, vol. 8324. Springer, San Diego, CA, USA, 152–167.

CAMPEOTTO, F., DOVIER, A., FIORETTO, F., AND PONTELLI, E. 2014. A GPU implementation of large neighborhood search for solving constraint optimization problems. In *ECAI 2014 - 21st European Conference on Artificial Intelligence - Including Prestigious Applications of Intelligent Systems (PAIS) 2014*, T. Schaub, G. Friedrich, and B. O'Sullivan, Eds. Frontiers in Artificial Intelligence and Applications, vol. 263. IOS Press, Prague, Czech Republic, 189–194.

CAMPEOTTO, F., DOVIER, A., AND PONTELLI, E. 2015. A Declarative Concurrent System for Protein Structure Prediction on GPU. *J. of Experimental & Theoretical Artificial Intelligence (JETAI)*. In press, on line from February, 24, 2015.

DAL PALÙ, A., DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2012. Exploiting unexploited computing resources for computational logics. In *Proc. of the 9th Italian Convention on Computational Logic*, F. A. Lisi, Ed. CEUR Workshop Proceedings, vol. 857. CEUR-WS.org, 74–88.

DAL PALÙ, A., DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2015. CUD@SAT: SAT Solving on GPUs. *J. of Experimental & Theoretical Artificial Intelligence (JETAI) 3*, 27, 293–316.

DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. 2009. GASP: answer set programming with lazy grounding. *Fundamenta Informaticae 96*, 3, 297–322.

DECHTER, R. 2003. *Constraint processing*. Elsevier Morgan Kaufmann.

FAGES, F. 1994. Consistency of clark's completion and existence of stable models. *Methods of Logic in Computer Science 1*, 1, 51–60.

FINKEL, R. A., MAREK, V. W., MOORE, N., AND TRUSZCZYNSKI, M. 2001. Computing stable models in parallel. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, 2001*.

FORMISANO, A. AND VELLA, F. 2014. On multiple learning schemata in conflict driven solvers. In *Proceedings of ICTCS 2014.*, S. Bistarelli and A. Formisano, Eds. CEUR Workshop Proceedings, vol. 1231. CEUR-WS.org, 133–146.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012a. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012b. Multi-threaded ASP solving with clasp. *TPLP 12*, 4-5, 525–545.

GELFOND, M. 2007. Answer sets. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier Science.

GOLDBERG, E. AND NOVIKOV, Y. 2007. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics 155*, 12, 1549–1561.

KAO, M. AND KLEIN, P. N. 1993. Towards overcoming the transitive-closure bottleneck: Efficient parallel algorithms for planar digraphs. *J. Computer and System Sciences 47*, 3, 459–500.

KHRONOS GROUP INC. 2015. *Open CL: The open standard for parallel programming of heterogeneous systems*. http://www.khronos.org.

KHULLER, S. AND VISHKIN, U. 1994. On the parallel complexity of digraph reachability. *Inf. Process. Lett. 52*, 5, 239–241.

LIN, F. AND ZHAO, Y. 2004. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence 157*, 1, 115–137.

LIU, L., PONTELLI, E., SON, T. C., AND TRUSZCZYNSKI, M. 2010. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence 174*, 3-4, 295–315.

MANOLIOS, P. AND ZHANG, Y. 2006. Implementing survey propagation on graphics processing units. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, 2006, Proceedings*, A. Biere and C. P. Gomes, Eds. LNCS, vol. 4121. Springer, 311–324.

MAREK, V. W. AND TRUSZCZYNSKI, M. 1998. Stable models and an alternative logic programming paradigm. *CoRR cs.LO/9809032*.

MARQUES SILVA, J. P. AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48*, 5, 506–521.

NICKOLLS, J. AND DALLY, W. 2010. The GPU computing era. *IEEE Micro 30*, 2, 56–69.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25*, 3-4, 241–273.

NVIDIA CORPORATION. 2015. *NVIDIA CUDA Zone*. https://developer.nvidia.com/cuda-zone.

PERRI, S., RICCA, F., AND SIRIANNI, M. 2013. Parallel instantiation of ASP programs: techniques and experiments. *TPLP 13*, 2, 253–278.

PONTELLI, E. AND EL-KHATIB, O. 2001. Exploiting vertical parallelism from answer set programs. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, Stanford, 2001*.

PONTELLI, E., LE, H. V., AND SON, T. C. 2010. An investigation in parallel execution of answer set programs on distributed memory platforms: Task sharing and dynamic scheduling. *Computer Languages, Systems & Structures 36*, 2, 158–202.

ROSSI, F., VAN BEEK, P., AND WALSH, T. 2006. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Inc. New York, NY, USA.

RUSSELL, S. J. AND NORVIG, P. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelliigence 138*, 1-2, 181–234.

SYRJÄNEN, T. AND NIEMELÄ, I. 2001. The smodels system. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, Proceedings*, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 434–438.

VELLA, F., DAL PALÙ, A., DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2013. CUD@ASP: Experimenting with GPGPUs in ASP solving. In *Proceedings of the 28th Italian Conference on Computational Logic, Catania, Italy.*, D. Cantone and M. Nicolosi Asmundo, Eds. CEUR Workshop Proceedings, vol. 1068. CEUR-WS.org, 163–177.