CUD@ASP: Experimenting with GPUs in ASP solving*

Flavio Vella[1], Alessandro Dal Palù[2], Agostino Dovier[3],
Andrea Formisano[1], and Enrico Pontelli[4]

[1] Dip. di Matematica e Informatica, Univ. di Perugia
[2] Dip. di Matematica, Univ. di Parma
[3] Dip. di Matematica e Informatica, Univ. di Udine
[4] Dept. of Computer Science, NMSU

Abstract. This paper illustrates the design and implementation of a prototype ASP solver that is capable of exploiting the parallelism offered by general purpose graphical processing units (GPGPUs). The solver is based on a basic conflict-driven search algorithm. The core of the solving process develops on the CPU, while most of the activities, such as literal selection, unit propagation, and conflict-analysis, are delegated to the GPU. Moreover, a deep non-deterministic search, involving a very large number of threads, is also delegated to the GPU. The initial results confirm the feasibility of the approach and the potential offered by GPUs in the context of ASP computations.

1 Introduction

Answer Set Programming (ASP) [22, 20] 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. ASP has offered novel and highly declarative solutions in several application areas, including intelligent agents, planning, software verification, complex 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, such as CLASP and DLV. 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 remains a challenging task, limiting the scope of applicability of ASP in certain domains (e.g., planning). In this work, we offer 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 (GPGPUs)*. Modern GPUs are multi-core platforms, offering massive levels of parallelism; vendors like NVIDIA have started

* Research partially supported by GNCS-13 project.

supporting the use of GPUs for applications different from graphical operations, providing dedicated APIs and development environments. Languages and language extensions like *OpenCL* [16] and *CUDA* [29] support the development of general purpose applications on GPUs, beyond the limitations of graphical APIs. 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.

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 research works, starting with seminal papers by Pontelli et al. and Finkel et al. [25, 9], and later continued in several other projects (e.g., [26, 12, 24]). 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; furthermore, the literature focused on parallelization on traditional multi-core or Beowulf architectures. These approaches are not applicable in the context of GPGPUs—the models of parallelization used on GPGPUs are deeply different (e.g., GPGPUs are designed to operate with large number of threads, operating in a synchronous way; GPGPUs have significantly more complex memory organizations, that have great impact on parallel performance) and existing parallel ASP models are not scalable on GPGPUs. Furthermore, our focus on this work is not primarily on search parallelism, but on parallelization of the various operations associated to unit propagation and management of nogoods.

The second line of research that supports the effort proposed in this paper is the recent developments in the area of GPGPUs for SAT solving and constraint programming. The work in [6] illustrates how to parallelize the search process employed by the DPLL procedure in solving a SAT problem on GPGPUs; the outcomes demonstrate the potential benefit of delegating to GPGPUs the tails of the branches of the search tree—an idea that we have also applied in the work presented in this paper. Several other proposals have appeared in the literature suggesting the use of GPGPUs to parallelize parts of the SAT solving process—e.g., the computation of variable heuristics [18]. The work presented in [4] provides a preliminary investigation of parallelization of constraint solving (applied to the specific domain of protein structure prediction) on GPGPUs. The work we performed in [4] provided inspiration for the ideas used in this paper to parallelize unit propagation and other procedures.

The main contribution of the research presented in this paper is the analysis of a state of the art algorithm for answer set computation (i.e., the algorithm underlying CLASP) to identify potential sources of parallelism that are suitable to the peculiar parallel architecture provided by CUDA.

2 Background

2.1 Answer Set Programming

Syntax. In this section we will briefly review the foundations of ASP, starting with its syntax. Let us consider a language composed of a set of propositional symbols (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 $p_i \in \mathcal{P}$.[5] Given a rule r of type (1), 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 $body^-(r) = \{p_{m+1}, \ldots, p_n\}$. We identify particular types of rules: a *constraint* is a rule of the form

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

while a *fact* is a rule of the form $p_0 \leftarrow$. A program Π is a collection of ASP rules. We will use the following notation: $atom(\Pi)$ denotes the set of all atoms present in Π, while $body_\Pi(p)$ denotes the set $\{body(r) \mid r \in \Pi,\ head(r) = p\}$.

Let Π be a program; its *positive dependence graph* $\mathcal{D}^+_\Pi = (V, E)$ is a directed graph satisfying the following properties:

- The set of nodes $V = atom(\Pi)$;
- $E = \{(p, q) \mid r \in \Pi, head(r) = p, q \in body^+(r)\}$.

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* (*non-tight*) if $loop(\Pi) = 0$ ($loop(\Pi) > 0$). A *strongly connected component (scc)* of $\mathcal{D}^+_\Pi$ is a maximal subgraph of X of $\mathcal{D}^+_\Pi$ such that there exists a path between each pair of nodes in X.

Semantics. The semantics of ASP programs is provided in terms of *answer sets*. Intuitively, an answer set is a minimal model of the program which supports each atom in the model—i.e., for each atom there is a rule in the program that has such atom in the head and whose body is satisfied by the model. Formally, a set of atoms M is an answer set of a program Π if M is the minimal model of the *reduct program* Π^M, where the reduct is obtained from Π as follows:

- remove from Π all rules r such that $M \cap body^-(r) \neq \emptyset$;
- remove all negated atoms from the remaining rules.

Π^M is a *definite program*, i.e., a set of rules that does not contain any occurrence of *not*. Definite programs are characterized by the fact that they admit a unique minimal model. Each answer set of a program Π is, in particular, a minimal model of Π.

Example 1. The following program Π has two answer sets: $\{a, c\}$ e $\{a, d\}$.

$$\Pi = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \neg a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Answer Set Computation. In the rest of this section, we provide a brief overview of techniques used in the computation of the answer sets of a program; the material presented is predominantly drawn from the implementation techniques used in CLASP [11, 10].

Several ASP solvers rely directly or indirectly on techniques drawn from the domain of SAT solving, properly extended to include procedures to determine minimality and stability of the models (these two procedures can be quickly performed in time linear in the number of occurrences of atoms in the program, namely $|\Pi|$)). Several ASP

[5] A rule that includes first-order atoms with variables is simply seen as a syntactic sugar for all its ground instances.

solvers (e.g., CMODELS [13]) rely on a translation of Π into a SAT problem and on the use of SAT solvers to determine putative answer sets. Other systems (e.g., CLASP) implement native ASP solvers, that combine search techniques with backjumping along with techniques drawn from the field of constraint programming [27].

The CLASP system relies on a search in the space 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 is implemented in such a way to guarantee that if a branch is successfully constructed, then the outcome is indeed 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* [7, 27]. Assignments and nogoods are sets of assigned atoms—i.e., entities of the form Tp (Fp) denoting that p has been assigned `true` (`false`). For assignments it is also required that for each atom p at most one between Tp and Fp is contained. Given an assignment A, we denote with $A^T = \{p \,|\, Tp \in A\}$ and $A^F = \{p \,|\, Fp \in A\}$. A is total if it assigns a truth value to every atom, otherwise it is partial. Given a (possibly partial) assignment A and a nogood δ, we say that δ is *violated* if $\delta \subseteq A$. In turn, a partial assignment 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 also used during deterministic propagation phases (a.k.a. *unit propagation*) to determine additional assignments. Given a nogood δ and a partial assignment A such that $\delta \setminus A = \{Fp\}$ ($\delta \setminus A = \{Tp\}$), then we can infer the need to add Tp (Fp) to A in order to avoid violation of δ. In the context of ASP computation, we distinguish two types of nogoods: *completion nogoods* [8], which are derived from Clark's completion of a logic program (we will denote with $\Delta_{\Pi_{cc}}$ the set of completion nogoods for the program Π), and *loop nogoods* [17], which are derived from the loop formula of Π (denoted by Λ_Π). Before proceeding with the formal definitions of these two classes of nogoods, let us review the two fundamental results associated to them (see [10]). Let Π be a program and A an assignment:

- If Π is a tight program then: $atom(\Pi) \cap A^T$ is an answer set of Π iff A satisfies all the nogoods in $\Delta_{\Pi_{cc}}$.
- If Π is a non-tight program, then: $atom(\Pi) \cap A^T$ is an answer set of Π iff A satisfies all the nogoods in $\Delta_{\Pi_{cc}} \cup \Lambda_\Pi$.

Let us now proceed in the formal definitions of nogoods. Let us start by recalling the notion of Clark completion of Π (Π):

$$\Pi_{cc} = \left\{\beta_r \leftrightarrow \bigwedge_{a \in body^+(r)} a \wedge \bigwedge_{b \in body^-(r)} \neg b \,|\, r \in \Pi\right\} \cup \left\{p \leftrightarrow \bigvee_{r \in body_\Pi(p)} \beta_r \,|\, p \in atom(\Pi)\right\} \quad (3)$$

Where β_r is a new variable, introduced for each rule $r \in \Pi$, logically equivalent to the body of r. Assignments need to deal with β_r variables, as well. The *completion nogoods* reflect the structure of the implications present in the definition of Π_{cc}. In particular:

- the implication present in the original rule $p \leftarrow body(r)$ implies the nogood $\{F\beta_r\} \cup \{Ta \,|\, a \in body^+(r)\} \cup \{Fb \,|\, b \in body^-(r)\}$.
- the implication in each rule also implies that the body should be false if any of its element is falsified, leading to the set of nogoods of the form: $\{T\beta_r, Fa\}$ for each $a \in body^+(r)$ and $\{T\beta_r, Tb\}$ for each $b \in body^-(r)$.
- the closure of an atom definition (as disjunction of the rule bodies supporting it) leads to a nogood expressing that the atom is true if any of its rule is true: $\{Fp, T\beta_r\}$ for each $r \in body_\Pi(p)$.
- similarly, the atom cannot be true if all its rules have a false body. This yields the nogood $\{Tp\} \cup \{F\beta_r \,|\, r \in body_\Pi(p)\}$.

$\Delta_{\Pi_{cc}}$ is the set of all the nogoods defined as above.

The *loop nogoods* derive instead from the need to capture loop formulae, thus avoiding cyclic support of truth. Let us provide some preliminary definitions. Given a set of atoms U, we define the *external bodies* of U (denoted by $EB_\Pi(U)$) as the set $\{\beta_r \,|\, r \in \Pi, body^+(r) \cap U = \emptyset\}$. Furthermore, let us define U to be an unfounded set with respect to an assignment A if, for each rule $r \in \Pi$, we have *(i)* $head(r) \notin U$, or *(ii)* $body(r)$ is falsified by A, or *(iii)* $body^+(r) \cap U \neq \emptyset$. The loop nogoods capture the fact that, for each unfounded set U, its elements have to be false. This is encoded by the following nogoods: for each set of atoms U and for each $p \in U$, we create the nogood $\{Tp\} \cup \{F\beta_r \mid \beta_r \in EB_\Pi(U)\}$. We denote with Λ_Π the set of all loop nogoods, and with Δ_Π the whole set of nogoods: $\Delta_\Pi = \Delta_{\Pi_{cc}} \cup \Lambda_\Pi$.

2.2 CUDA

Our proposal focuses on exploring the use GPGPU parallelism in ASP solving. GPGPU is a general term indicating the use of the multicores available within modern graphical processing units (GPUs) for general purpose parallel computing. NVIDIA is one of the pioneering manufacturers in promoting GPGPU computing, especially thanks to its *Computing Unified Device Architecture (CUDA)* [29]. The underlying conceptual model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT),* a variant of the SIMD model, where, in general, 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 1.

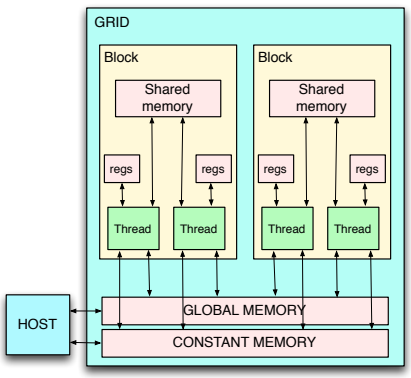

Fig. 1: CUDA Logical Architecture

Different NVIDIA GPUs are distinguished by the number of cores, their organization, and the amount of memory available. The GPU is composed of a series of *Streaming MultiProcessors (SMs);* the number of SMs depends on the specific characteristics of each family of GPU—e.g., the Fermi architecture provides 16 SMs. In turn, each SM contains a collection of computing cores; 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 both on-chip memory (used for thread registers and shared memory—defined later) and on-chip memory (used for L2 cache, global memory and constant memory). Notice that the architecture of the GPU also determines both the *GPU Clock* and the *Memory Clock* rates. A logical view of computations is introduced by CUDA, in order to define abstract parallel work and to schedule it among different hardware configurations (see Figure 1). A typical CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the *host*) and parts meant for parallel execution on the GPU (referred as the *device*). A parallel computation is described by a collection of *kernels*—each kernel is a function to be executed by several threads.

The host program contains all instructions to initialize the data in GPUs, to define the threads number and to manage the kernel. Instead, a kernel is a set of instruction performed in GPUs across a set of concurrent threads. The programmer or compiler organizes these threads in thread *blocks* and *grids* of thread blocks. A grid is an array of thread blocks that execute the same kernel, read data input from global memory, write results to global memory. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block. When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity; the kernel is executed in N blocks, each consisting of M threads. The threads in the same block can share data, using shared high-throughput on-chip memory; on the other hand, the threads belonging to different blocks can only share data through global memory. Thus, the block size allows the programmer to define the granularity of threads cooperation. Figure 1 shows the CUDA threads hierarchy [23].

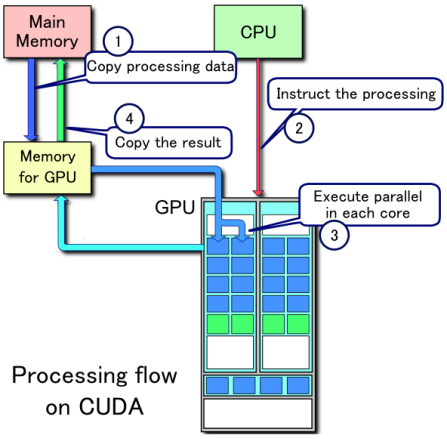

Fig. 2: Generic workflow in CUDA

CUDA provides an API to interact with GPU and C for CUDA, an extension of C language to define kernels. Referring to Figure 2, a typical CUDA application can be summarized as follow:

Memory data allocation and transfer: The data before being processed by kernels, must be allocated and transferred to Global memory. The CUDA API supports this operations through the functions `cudaMalloc()` and `cudaMemcpy()`. The call `cudaMalloc()` allows the programmer to allocate the space needed to store the data while the call `cudaMemcpy()` transfers the data from the memory of the host to the space previously allocated in Global Memory, or *vice versa*. The transfer rate is dependent on the bus bandwidth where the Graphics Card is physically connected.

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)` where `__global__` is the qualifier that shows to the compiler that the next statement is a kernel code.
Kernels execution: A kernel can be launched from the host program using a new:
`kernelName <<< GridDim, ThreadsPerBlock >>> (Actual Arguments)`
execution configuration syntax where `kernelName` is the specified name in kernel function prototype, `GridDim` is the number of blocks of the grid and `ThreadsPerBlock` specifies the number of threads in each block. Finally, the `Actual Arguments` are typically pointer variables, referring to the previously allocated data in Global Memory.
Data retrieval: After the execution of the kernel, the host needs to retrieve the data—representing results of the kernel. This is performed with another transfer operation from Global Memory to Host Memory, using the function `cudaMemcpy()`.

3 Design of an conflict-based CUDA ASP Solver

In this section, we will present the `CUD@ASP` procedure. This procedure is based on the `CDNL-ASP` procedure adopted in the CLASP system [11, 10]. The procedure assumes that the input is a ground ASP program. The novelty of `CUD@ASP` is the off-loading of several time consuming operations to the GPU—with particular focus on conflict analysis, exploration of the set of possible assignments and execution of the phases of unit-propagation. The rest of this section is organized as follows: we will start with an overview of the serial structure of the `CUD@ASP` procedure (Subsection 3.1). In the successive subsections, we will illustrate the parallel versions of the key procedures used in `CUD@ASP`: literal selection (Subsection 3.2), nogoods analysis (Subsection 3.3), unit propagation (Subsection 3.4), conflict analysis (Subsection 3.5), and analysis of stability (Subsection 3.7). In addition, we illustrate a method to use the GPU to handle the search process in the tail part of the search tree (Subsection 3.6).

3.1 The General `CUD@ASP` Procedure

The overall `CUD@ASP` procedure is summarized in Algorithm 3.2. The procedures that appear underlined in the algorithm are those that are delegated to the GPU for parallel execution. The algorithm makes use of the following notation. The input (ground) program is denoted by Π; Π_{cc} denotes the completion of Π (eq. 3). The overall set of nogoods is denoted by Δ_Π, composed of the completion nogoods and the loop nogoods. For each program atom p, the notation $\overline{p}$ represents the atom with a truth value assigned; $\neg\overline{p}$ denotes, instead, the complement truth value with respect to $\overline{p}$.

Lines 1–5 of Algorithm 3.2 represent the initialization phase of the ASP computation. In particular, the `Parsing` procedure (Line 5) is in charge of computing the completion of Π and extracting the nogoods. The set A will keep track of the atoms that have already been assigned a truth value. It is initialized to the empty set in Line 1 and updated by the `Selection` procedure at Line 22. Two variables ($current\_dl$ and k) are introduced to support the rest of the computation. In particular, the variable $current\_dl$ represents the *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. Line 6 invokes the procedure `StronglyConnectedComponent`, which

Algorithm 3.2 CUD@ASP

Input Π ground ASP program
Output An answer set, or null

```
 1: A := ∅                                                    ▷ Atoms assignment
 2: Δ_Π := ∅                                                  ▷ Nogoods
 3: current_dl := 0                                           ▷ Current Decision Level
 4: k := 32                                                   ▷ Threshold for Exhaustive Procedure
 5: (Δ_Π, k, Π_cc) := Parsing(Π)                              ▷ Initialize Δ_Π as Δ_{Π_cc}
 6: scc := StronglyConnectedComponent(Π)
 7: loop
 8:     Violation := NoGoodCheck(A, Δ_Π)
 9:     if (Violation is true) ∧ (current_dl = 0) then return no answer set
10:     end if
11:     if Violation is true then
12:         (current_dl, δ) = ConflictAnalysis(Δ_Π, A)
13:         Δ_Π = Δ_Π ∪ {δ}
14:         A := A \ {p̄ ∈ A | current_dl < dl(p̄)}
15:     else
16:         if ∃ δ ∈ Δ_Π such that δ \ A = {p̄} and p̄ ∉ A then
17:             A := UnitPropagation(A, Δ_Π)
18:         end if
19:     end if
20:     if There are atoms not assigned then
21:         if Number of atoms to assign > k then
22:             p̄ := Selection(Π_cc, A)
23:             current_dl := current_dl + 1
24:             dl(p̄) := current_dl
25:             A := A ∪ {p̄}
26:         else          ▷ At most k unassigned atoms: Non-deterministic GPU computation
27:             if There is a successful thread for Exhaustive(A) then
28:                 for each successful thread returning A := Exhaustive(A) do
29:                     if StableTest(A, Π_cc) is true then return A^T ∩ atom(Π)
30:                     end if
31:                 end for
32:             end if
33:         end if
34:     else return A^T ∩ atom(Π)
35:     end if
36: end loop
```

determines the positive dependence graph and its strongly connected components; in absence of loops, the program Π is tight, thus not requiring the use of loop nogoods (Λ_Π). We have implemented the classical Tarjan's algorithm, running in $O(n+e)$, on CPU (where n and e are the numbers of nodes and edges, respectively). The loop in Lines 7–36 represents the core of the computation. It alternates the process of testing consistency and propagating assignments (through the nogoods), and of guessing a possible assignment to atoms that are still undefined. Each cycle starts with a call to the

procedure `NoGoodCheck` (Line 8)—which, given a partial assignment A, validates whether all the nogoods in Δ_Π are still satisfied. If a violation is detected, then the procedure `ConflictAnalysis` is used to determine the decision level causing the nogood violation, backtrack to such point in the search tree, and generate an additional nogood to prune that branch of the search space (Lines 11–14). If $\overline{p}$ is the assignment at the decision level determined by `ConflictAnalysis`, then the nogood will prompt the unit propagation process to explore the branch starting with the truth assignment $\neg\overline{p}$ (thus ensuring completeness of the computation [10]).

If the `ConflictAnalysis` procedure does not detect nogood violations, then the procedure might be in one of the following situations:

- If there is a nogood that is completely covered by A except for one element $\overline{p}$, then the `UnitPropagation` procedure is called to determine assignments that are implied by the nogoods (starting with the assignment $\neg\overline{p}$) (Lines 16–17). Note that this procedure does not modify the decision level. In the case of non-tight programs, the `UnitPropagation` procedure will also execute a subroutine in charge of validating the loop nogoods.
- If there are atoms left to assign (Line 20), then additional selections will need to be performed. We distinguish two possibilities. If the number of unassigned atoms is larger than a threshold k, then one of them, say $\overline{p}$, is selected and the current decision level is recorded (by setting the value of the variable $dl(\overline{p})$—see Line 24). The `Selection` procedure is in charge for selecting a literal. The assignment is extended accordingly and the current decision level is increased (Lines 23–25). If the number of unassigned atoms is small, then a specialized parallel procedure (`Exhaustive`) systematically explores all the possible missing assignments. For each possible assignment of the remaining atoms, the procedure `StableTest` validates that all nogoods are satisfied and that the overall assignment A is stable (necessary test in the case of non-tight programs). This is described in Lines 27–32.

3.2 `Selection` Procedure

The purpose of this procedure is to determine an unassigned atom in the program and a truth value for it. A number of heuristic strategies have been studied to determine atom and assignment, often derived from analogous strategies developed in the context of SAT solving or constraint solving [27, 2]. As soon as an atom has been selected, it is necessary to assign a truth value to it. A traditional strategy [10] consists of assigning at the beginning the value `true` to bodies of rules, while atoms are initially assigned `false`—aiming at maximizing the number of resulting implications.

There is no an optimal strategy for all problems, of course. In the current implementation, we provide three selection strategies: the most frequently occurring literal strategy which selects the atom that appears in the largest number of nogoods (that aims at determining violations as soon as possible or to lead to early propagations through the nogoods), the leftmost-first strategy (which selects the first unassigned atom found), and the *Jeroslow-Wang* strategy (also based on the frequency of occurrence of an atom, but placing a greater value on smaller nogoods). All the three strategies are implemented by allowing kernels on the GPU to concurrently compute the rank of each atom; these rankings are re-evaluated at each backjump.

Algorithm 3.3 `NoGoodCheck` ▷ Kernel executed by thread i

Input A, $\Delta_\Pi = \{\delta_1, \ldots, \delta_m\}$ ▷ An assignment A and a set of nogoods Δ_Π
Output True or False

```
 1: if i ≤ m then
 2:     state := 0
 3:     covered := 0
 4:     Atom to propagate := NULL
 5:     for all p̄ ∈ δ_i do
 6:         if ¬p̄ ∈ A then state := 1
 7:         else if p̄ ∈ A then covered := covered + 1
 8:         else Atom to propagate := p̄
 9:         end if
10:     end for
11:     if covered = |δ_i| then return Violation := True
12:     else if covered = |δ_i| − 1 and state = 0 then
13:         Make Atom to propagate global
14:     end if
15:     return Violation := False
16: end if
```

3.3 `NoGoodCheck` Procedure

The `NoGoodCheck` procedure (see Algorithm 3.3) is primarily used to verify whether the current partial assignment A violates any of the nogoods in a given set Δ_Π. The procedure plays also the additional rôle of identifying opportunities for unit propagation—i.e., recognizing nogoods δ such that $\delta \setminus A = \{\overline{p}\}$ and $\neg\overline{p} \notin A$. In this case, the element p will be the target of a successive unit propagation phase.

The pseudocode in Algorithm 3.3 describes a CUDA kernel (i.e., running on GPU) implementing the `NoGoodCheck`. Each thread handles one of the nogoods in Δ_Π and performs a linear scan of its assigned atoms (Lines 5–10). The local flag `state` keeps track of whether the nogood is satisfied by the assignment (`state` equal to 1). The counter `covered` keeps track of how many elements of δ_i have already been found in A. The condition of `state` equal to zero and the `covered` counter equal to the size of the nogood implies that the nogood is violated by A. The first thread to detect a violation will communicate it to the host by setting a variable (`Violation`—Line 11) in global memory (used in Lines 9 and 11 of the general `CUD@ASP` procedure).

Lines 12–13 implement the second functionality of the `NoGoodCheck` procedure—by identifying and making global the single element of the nogood that is not covered by the A assignment. Note that the identification of the element `Atom to Propagate` can be conveniently performed in `NoGoodCheck` since the procedure is already performing the scanning of the nogood to check its validity.

3.4 `UnitPropagation` Procedure

The `UnitPropagation` procedure is performed only if the `NoGoodCheck` has detected no violations and has exposed at least one atom for propagation (as in Lines

12–13 of Algorithm 3.3). `UnitPropagation` is implemented as a CUDA kernel—which allows us to distribute the different nogoods among threads, each in charge of extending the partial assignment A with one additional assignment. The procedure is iterated until a fixpoint is reached. The extension of A is an immediate consequence of the work done in `NoGoodCheck`: if the check of a nogood δ_i identifies $\overline{p}$ as the only element in δ_i not covered by A (i.e., $\{\overline{p}, \neg\overline{p}\} \cap A = \emptyset$), then A is extended as $A := A \cup \{\neg\overline{p}\}$.

If the program Π is non-tight, then the `UnitPropagation` procedure includes an additional phase aimed at performing the computation of the unfounded sets determined by the partial assignment A and the corresponding loop nogoods Λ_Π. This process is implemented by the procedure `UnfoundedSetCheck` and follows the general structure of the analogous procedure used in the implementation of CLASP [10]. This procedure performs an analysis of the strongly connected components of the positive dependence graph $\mathcal{D}^+_\Pi$ (already computed at the beginning of the computation of `CUD@ASP`—Line 6). For each $p \in atoms(\Pi)$, $scc(p)$ denotes the set of atoms that belong to the same strongly connected component as p. An atom p is said to be cyclic if there exists a rule $r \in \Pi$ such that: $head(r) \in scc(p)$ and $body^+(r) \cap scc(p) \neq \emptyset$, otherwise p is *acyclic*. Cyclic atoms are the core of the search for unfounded sets—since they are the only ones that can appear in the unfounded loops. Cyclic atoms along with the knowledge of elements assigned by A allow the computation of unfounded sets, as discussed in [17, 10]. In the current implementation `UnfoundedSetCheck` runs on the host. Some parts are inherently parallelizable (e.g., the computation of the external-support, or a splitting to different threads of the analysis of each scc component)—their execution on the device is work in progress.

3.5 `ConflictAnalysis` Procedure

The `ConflictAnalysis` procedure is used to resolve a conflict detected by the `NoGoodCheck` by identifying a level dl and 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 the Algorithm 3.2 [28, 27]. In addition to this, the procedure produces a new nogood to be added to the nogoods set, in order to prevent the same assignments in future. This procedure is implemented by a sequence of kernels, and it is executed after some nogood violations have been detected by `NoGoodCheck`. This procedure works as follows:

- Each thread is assigned to a unique nogood (δ).
- The thread determines the last two assigned literals in δ, say $\ell_M(\delta)$ and $\ell_m(\delta)$. The two (not necessarily distinct) decision levels of these assignments are stored in $dl_M(\delta) = dl(\ell_M(\delta))$ and $dl_m(\delta) = dl(\ell_m(\delta))$, respectively.
- The thread verifies whether δ is violated.
- Then, the violated nogood $\overline{\delta}$ with lowest value of dl_M is determined.

At this point, a nogood learning procedure is activated. A kernel function (again, one thread for each existing nogood) determines each nogood ε, such that: *(a)* $\neg\ell_M(\overline{\delta}) \in \varepsilon$ and *(b)* $\varepsilon \setminus \{\neg\ell_M(\overline{\delta})\} \subseteq A$. Heuristic functions (see, e.g., [1]) can be applied to select one of these ε. Currently, the smallest one is selected in order to generate small new

nogoods—as future work, we will consider all the set of these nogoods. The next step performs a sequence of steps, by repeatedly setting $\overline{\delta} := (\varepsilon \backslash \{\neg \ell_M(\overline{\delta})\}) \cup (\overline{\delta} \backslash \{\ell_M(\overline{\delta})\})$ and coherently updating the values of $dl_M(\overline{\delta})$ and $dl_m(\overline{\delta})$, until $dl_M(\overline{\delta}) \neq dl_m(\overline{\delta})$. This procedure ends with the identification of a unique implication point (UIP [21]) that determines the lower decision level/literal among those causing the detected conflicts. We use such value for backjumping (Line 14 of Algorithm 3.2). The last nogood obtained in this manner is also added to the set of nogoods.

3.6 `Exhaustive` Procedure

GPU are typically employed for data parallelism. However, as shown in [6], when the size of the problem is manageable, it is possible to use them for massive search parallelism. We have developed the `Exhaustive` procedure for this task. It is called when at most k atoms remains undecided—where k is a parameter that can be set by the user (by default, $k = 32$). The nogood set is simplified using the current assignment (this is done in parallel by a kernel that assigns each nogood to a thread). This simplified sets will be then processed by a second kernel with 2^k threads, that non-deterministically explores all of the possible assignments. Each thread verifies that the assignments do not violate the nogoods set. If this happens, in case of a tight program, we have found an answer set. Otherwise the `StableTest` procedure (Sect. 3.7) is launched (Lines 27–28 of Algorithm 3.2). The efficiency of this procedure is obtained by a careful use of low-level data-structures. For example, the Boolean assignment of 32 atoms is stored in a single integer variable. Similarly, the nogood representation is stored using bit-strings, and violation control is managed by low-level bit operations.

3.7 `StableTest` Procedure

In order to verify whether an assignment found by the `Exhaustive` procedure is a stable model, we have implemented a GPU kernel that behaves as follows:

- It computes the reduct of the program: each thread takes care of an individual rule; as result, some threads may become inactive due to rule elimination, threads dealing with rules with all negative literals not in the model simply ignore them, while all other threads are idle.
- A computation of the minimum fixpoint is performed. Each thread handles one rule (internally modified by the first step above) and, if the body is satisfied, updates the sets of derived atoms. Once a rule is triggered, it becomes inactive, speeding-up the consecutive computations.
- When a fixpoint is reached, the computed and the guessed models are compared.

4 Concluding discussion

We have reported on our working project of developing an ASP solver running (partially) on GPGPUs. We implemented a working prototypical solver. The first results in experimenting with different GPU architectures are encouraging. Table 1 shows an

excerpt of the results obtained on some instances (taken from the Second ASP Competition). The differences between the performance obtained by exploiting different GPUs are evident and indicates the strong potential for enhanced performance and the scalability of the approach.

Table 2 reports on the performance of different serial ASP solvers, on the same collection of instances. Far from being a deep and fair comparison of these solvers against the GPU-based prototype, these results show that even at this stage of its development, the parallel prototype can compete, in some cases, with the existing and highly optimized serial solvers. Notice that the GPU-based prototype does not benefit from a number of refined heuristics and search/decision strategies exploited, for instance, by the state of the art solver CLASP.

It should be noticed that, in order to profitably exploit in full the computational power of the GPUs, one has to carefully tune its parallel application w.r.t. the characteristics of the specific device at hand. The architectural features and characteristics of the specific GPU family has to be carefully taken into account. Moreover, even considering a given GPU, different options can be adopted both in partitioning tasks among threads/warps and in allocating/transferring data on the device's memory. Clearly, such choices sensibly affect the performance of the whole application. This can be better explained by considering Table 3. It shows the performance obtained by three versions of the GPU-based solver, differing in the way the device's global memory is used. Apart from the default allocation mentioned in Sect. 2.2, CUDA provides two other basic kind of memory allocation. A first possibility uses *page-locking* to speed up address resolution. *Mapped* allocation allows one to map a portion of host memory into the device global memory. In this way the data transfer between host and device is implicitly ensured by the system and explicit memory transfers (by means of the function `cudaMemcpy()`) can be avoided. The first column of Table 3 shows the performance of a version of the prototype that allocates all data by using *mapped* memory. The behavior of a faster version of the solver which exploits *page-locking* to deal with the main data structures (essentially those representing the set of nogoods), is shown in the second column. Clearly, this approach requires additional programming effort (in optimizing and keeping track of memory transfers). Even better performance has been achieved by a third version of the solver that adopts *page-locking* to allocate all data structures, only on the device. This solution may appear, in some sense, unappealing, because it imposes to implement on the device also some intrinsically-serial functionalities. Even if these functions cannot fully exploit the parallelism of the cores, considerable advantage is achieved by avoiding most of the memory transfer between host and device.

In this work we made initial steps towards the creation of a GPU-based ASP-solver; however, further effort is needed to improve the solver. In particular, some procedures need to be optimized in order to take greater advantage from the high data-/task-parallelism offered by GPGPUs and the different types of available memories. Moreover, some parts of the solver currently running on the host, should be replaced by suitable parallel counterparts (examples are the computation of the strongly connected components of the dependence graph and the computation of the unfounded sets). We plan to develop the stability test that avoids analyzing the whole program and the implementation of the `NoGoodCheck` that makes use of watched literals.

Instance	GT520	GT640	GTX580
channelRoute_3	5.44	1.73	0.37
knights_11_11	0.70	0.23	0.06
knights_13_13	1.70	0.51	0.12
knights_15_15	1.71	0.51	0.12
knights_17_17	2.40	0.69	0.16
knights_20_20	8.57	2.34	0.46
labyrinth.0.5	0.08	0.08	0.05
schur_4_41	0.24	0.16	0.07
schur_4_42	0.31	0.20	0.07

Table 1. Results obtained with three different Nvidia GeForce GPUs: GT520 (48 cores, capability 2.0, GPU clock 1.62 GHz, memory clock rate 0.50 GHz, global memory 1GB), GT640 (384 cores, capability 3.0, GPU clock 0.90 GHz, memory clock rate 0.89 GHz, global memory 2GB), GTX580 (512 cores, capability 2.0, GPU clock 1.50 GHz, memory clock rate 2.00 GHz, global memory 1.5GB). The timing is in seconds.

References

[1] C. Anger, M. Gebser, and T. Schaub. Approaching the Core of Unfounded Sets. Proceedings of the International Workshop on Nonmonotonic Reasoning. 2006.

[2] A. Biere. *Handbook of Satisfiability*, IOS Press, 2009.

[3] H. Blair and A. Walker. *Towards a theory of declarative knowledge*. IBM Watson Research Center, 1986.

[4] F. Campeotto, A. Dovier, and E. Pontelli. Protein Structure Prediction on GPU: an experimental report. *Proc. of RCRA*, Rome, June 2013.

[5] K. Clark. Negation as Failure. *Logic and Databases*, Morgan Kaufmann, 1978.

[6] A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli. Exploiting Unexploited Computing Resources for Computational Logics. *Proc. of CILC*, CEUR, vol 857, 2012.

[7] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[8] F. Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[9] R. Finkel et al. Computing Stable Models in Parallel. *Answer Set Programming*, AAAI Spring Symposium, 2001.

[10] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven Answer Set Solving: From Theory to Practice. *Artificial Intelligence* 187: 52–89, 2012.

[11] M. Gebser et al. *Answer Set Solving in Practice.* Morgan & Claypool, 2012.

[12] M. Gebser et al. Multi-Threaded ASP Solving with CLASP. *TPLP*, 12(4-5), 2012.

[13] E. Giunchiglia et al. Answer Set Programming Based on Propositional Satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[14] J. Herbrand. *Recherches sur la théorie de la démonstration*. Doctoral Dissertation, Univ. of Paris, 1930.

[15] R. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Math and AI*, 1:167-187, 1990.

[16] Khronos Group Inc. OpenCL Reference Pages. http://www.khronos.org, 2011.

[17] F. Lin and Y. Zhao. Assat: Computing Answer Sets of a Logic Program by SAT Solvers. *Artificial Intelligence*, 157(1):115–137, 2004.

[18] P. Manolios and Y. Zhang. Implementing Survey Propagation on Graphics Processing Units. *SAT*, Springer Verlag, 2006.

Instance	SMODELS	CMODELS	CLASP-None	CLASP	GTX580
channelRoute_3	2.08	1.42	69.27	0.24	0.37
knights_11_11	0.34	0.11	0.03	0.03	0.06
knights_13_13	1.12	0.21	0.06	0.06	0.12
knights_15_15	1.12	0.24	0.05	0.07	0.12
knights_17_17	0.91	1.99	0.05	0.06	0.16
knights_20_20	9.61	3.85	0.22	0.20	0.46
labyrinth.0.5	0.02	0.01	0.01	0.01	0.05
schur_4_41	0.05	0.70	0.02	0.02	0.07
schur_4_42	0.07	0.60	0.02	0.05	0.07

Table 2. Results obtained with different solvers. All experiments where run on the same machine (host: QuadCore Intel i7 CPU, 2.93GHz, 4GB RAM; device GTX580). Serial solvers: SMODELS v. 2.34; CMODELS v. 3.85 exploiting minisat; CLASP v. 2.1.0. The column 'CLASP-None' shows results obtained with CLASP by inhibiting its decision heuristics. This makes its selection strategy analogous to the one used in our implementation. The timing is in seconds.

Instance	All data mapped	Δ_Π page-locked	All data page-locked
knights_11_11	0.87	0.16	0.06
knights_13_13	2.50	0.34	0.12
knights_15_15	2.49	0.35	0.12
knights_17_17	3.60	0.50	0.16
knights_20_20	14.14	1.60	0.46
labyrinth.0.5	0.82	0.03	0.05
schur_4_41	19.71	0.09	0.07
schur_4_42	24.75	0.12	0.07

Table 3. Results obtained with GTX580 with different use of memory resources.

[19] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM (JACM)*, 38(3):587–618, 1991.

[20] W. Marek and M. Truszczyński. Stable Models as an Alternative Programming Paradigm. *The Logic Programming Paradigm*, Springer Verlag, 1999.

[21] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48:506-521, 1999.

[22] I. Niemela. Logic Programming with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Math and AI*, 25:241-273, 1999.

[23] J. Nickolls and W.J. Dally. The GPU Computing Era. In *IEEE Micro*, 30(2):56-59, 2010.

[24] S. Perri, F. Ricca, and M. Sirianni. Parallel Instantiation of ASP Programs: Techniques and Experiments. *TPLP*, 13(2), 2013.

[25] E. Pontelli and O. El-Khatib. Exploiting Vertical Parallelism from Answer Set Programs. *Answer Set Programming*, AAAI Spring Symposium, 2001.

[26] E. Pontelli, H. Le and T. Son. An Investigation in Parallel Execution of ASP on Distribute Memory Platforms. *Computer Languages, Systems & Structures*, 36(2):158-202, 2010.

[27] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*, Elsevier, 2006.

[28] S. Russell et al. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[29] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2011.

[30] A. Van Gelder et al. The Well-founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):619–649, 1991.