GPU Parallelism for SAT Solving Heuristics*

Michele Collevati[1], Agostino Dovier[1,2] and Andrea Formisano[1,2,*]

[1]CLPLab - DMIF - Università di Udine, via delle Scienze 206, 33100 Udine, Italy
[2]GNCS-INdAM, piazzale Aldo Moro 5, 00185 Roma, Italy

Abstract

Modern SAT solvers employ a number of smart techniques and strategies to achieve maximum efficiency in solving the Boolean Satisfiability problem. Among all components of a solver, the branching heuristics plays a crucial role in affecting the performance of the entire solver. Traditionally, the main branching heuristics that have appeared in the literature have been classified as *look-back* heuristics or *look-ahead* heuristics. As SAT technology has evolved, the former have become more and more preferable, for their demand for less computational effort.

Graphics Processor Units (GPUs) are massively parallel devices that have spread enormously over the past few decades and offer great computing power at a relatively low cost. We describe how to exploit such computational power to efficiently implement look-ahead heuristics. Our aim is to "rehabilitate" these heuristics, by showing their effectiveness in the contest of a parallel SAT solver.

Keywords

SAT Solving, Branching Heuristics, GPU parallelism

1. Introduction

The central point of either DPLL [1] or CDCL [2] SAT solvers is the choice of the successive variable to be assigned (*variable selection heuristics*) and the choice of the Boolean value to be attempted first (*polarity selection heuristics*). The algorithms for implementing the two choices are called *branching heuristics*, and, apart from the naive ones (e.g., leftmost variable, random choice, etc.), they can be classified as *look-back* heuristics or *look-ahead* heuristics. The former are, in general, easier to implement, since it is sufficient to collect and maintain minimal information about the evolution of the computation. Look-ahead heuristics require a (partial) exploration of the "future" of the computation in order to determine the potential impact of alternative choices the solver can make at a choice point and this can be computationally expensive. This is a reason why look-ahead heuristics have largely been abandoned in modern solvers, in favor of the "lighter" (and, somehow, possibly coarser) look-back heuristics.

CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy

*Research partially supported by Fondazione Friuli/Università di Udine project on *Artificial Intelligence for Human Robot Collaboration* and by projects INDAM GNCS-2020 *NoRMA* and INDAM GNCS-2022 InSANE (CUP_E55F22000270001).

*Corresponding author.
michele.collevati@protonmail.com (M. Collevati); agostino.dovier@uniud.it (A. Dovier); andrea.formisano@uniud.it (A. Formisano)
0000-0001-7958-7841 (M. Collevati); 0000-0003-2052-8593 (A. Dovier); 0000-0002-6755-9314 (A. Formisano)

GPU manufacturer NVIDIA, through its platform called *Computing Unified Device Architecture (CUDA)* [3], is a leading pioneer in GPU-computing. CUDA, unveiled in 2006, is a general-purpose parallel computing platform and programming model that leverages the parallel computing engine of NVIDIA GPUs. It can be programmed in C or C++ and it enables the development of applications that scale their parallelism transparently and take advantage of the growing number of CPU and GPU cores. Although initially GPU were used for graphical purposes, e.g., video games, they are nowadays widely used in Deep Learning computation. A stream of works trying to exploit them for SAT/ASP solving exists [4, 5, 6, 7].

In this paper, we describe how to exploit the computational power of GPUs by developing a CUDA C library that implements the look-ahead heuristics. In a sense, our aim is to "rehabilitate" these heuristics, by showing their effectiveness in the contest of a parallel SAT solver. In Section 2 we introduce the main notions and notation used in the paper. Section 3 describes the main ideas behind the GPU implementation of the look-ahead heuristics. In Section 4 we report on the experiments made using our implementation with a DPLL and a CDCL solver. Conclusions are drawn in Section 5.

2. Background

Let $\mathcal{V}$ be a (denumerable) set of variables. If $x \in \mathcal{V}$ then x and $\neg x$ are said *literals*. A disjunction $\omega = (\ell_1 \vee \cdots \vee \ell_k)$ of literals is said a *clause*. If $k = 1$ the clause ω is said *unit clause* (or, simply, *unit*). A Boolean formula Φ in *Conjunctive Normal Form (CNF)* is a conjunction $(\omega_1 \wedge \cdots \wedge \omega_h)$ of clauses. As common, for denotational convenience, we might refer to Φ as a set of clauses and, similarly, to a clause ω as a set of literals.

A *(partial) Boolean assignment* σ is a mapping from $X \subseteq \mathcal{V}$ to $\{\mathsf{false}, \mathsf{true}\}$. An assignment can be applied to literals, clauses, and formulas, and evaluated using the classical semantics of propositional connectives $\neg, \vee, \wedge$. In particular, for any clause ω, $\sigma(\omega) = \mathsf{true}$ if and only if $\sigma(\ell) = \mathsf{true}$ for some $\ell \in \omega$. In such case, we say that ω is *satisfied* by σ. If $\sigma(\ell) = \mathsf{false}$ for each $\ell \in \omega$, then $\sigma(\omega) = \mathsf{false}$. Given a set of clauses Φ (i.e., a CNF Boolean formula), an assignment σ is a *solution* for Φ if σ satisfies all $\omega \in \Phi$. The *Boolean Satisfiability (SAT)* problem is the problem of establishing whether a solution exists for a given Φ.

An assignment σ may be partial. Namely, it might be the case that $\sigma(\ell)$ is not defined for some $\ell \in \mathcal{V}$. In case some literals in a clause ω are not assigned by σ, and ω is not satisfied by σ, the clause is said *unresolved* (w.r.t. σ). An unresolved clause ω with only one undefined literal is said *unit* (w.r.t. σ).

2.1. SAT Solving

The relevance of SAT for the $\mathcal{P}$ *versus* $\mathcal{NP}$ problem is clear since the seminal papers by Cook and Levin [8, 9]. However, the research of an automatic procedure capable of solving concrete instances of SAT was already one of the most important research area of theorem proving. In this context, a milestone was posed by Davis and Putnam [10] developing a procedure later refined for reducing space occupation by their co-authors Logemann and Loveland [1]. The algorithm, known as *DPLL*, combines three stages:

1. *(unit) propagation*: deterministically infer values for variables under a given partial assignment σ: whenever a clause ω is unit w.r.t. σ, extend σ so as to satisfy ω;
2. *choice*: non-deterministically choose a not yet assigned variable x, assign it a value among false and true, and extend the current partial assignment accordingly;
3. *backtracking*: if a failure is reached because no solutions were found under the current assignment, backtrack the last choice made in assigning a variable.

The simplest possible solver starts from the empty assignment (all variables are unassigned) and alternates propagation and choice steps. At each moment in time, the number of active choices performed is the *decision level* currently reached in the search for a solution. The search proceeds until either a solution is found (i.e., an assignment satisfying all clauses) or a clause gets falsified by the current (partial) assignment. In this case the computation backtracks and the decision level is decreased (undoing the effects of the last choice). Hence, the solver proceeds by visiting a tree-shaped search space and the decision level is the depth currently reached in the search tree.

Implementing a DPLL solver requires the selection of the algorithms to perform step (2). Concretely, one has to choose two heuristics to be used in *choice points* to select

- the variable to be assigned, called *Variable Selection Heuristics (VSH)*, and
- the truth value to be attempted first, called *Polarity Selection Heuristics (PSH)*.

A second family of SAT solvers extends the idea of DPLL by analyzing the reasons why an assignment has lead the search to a failure. Solvers of this family are called *Conflict-Driven Clause-Learning (CDCL)* solvers. These solvers proceed as DPLL until a failure is detected. Then, a step called *conflict analysis* is performed to detect a reason for the failure, namely, a set of variable assignments (made by the choice steps (2)) that conjunctively prevent the satisfaction of some clauses of the input formula Φ. This set of variable assignments identifies a new clause that can be *learned* and added to Φ. The rational is that any learned clause ω is a logical consequence of Φ. Hence, if σ is a solution of Φ, it is also a solution of $\Phi \wedge \omega$. After a new clause is learned, the solver *backjumps* to a decision level preceding (at least some of) conflicting choices (undoing their effects on the current assignment). Each learned clause is expected to speed up the subsequent search because it prevents the solver from making the same failing set of assignments again. Here, usually, unit propagation enters into play: whenever all but one of the assignments of such set are done, the presence of the learned clause forces a different value selection for the remaining assignment. Clearly, "short" learned clauses are more effective in speeding up the search by reducing the search space. We refer the reader to [2] for a detailed formal description of CDCL solvers.

The branching heuristics can be partitioned into two families:

1. *look-back* heuristics, which rank variables on the basis of the computation performed till the choice point;
2. *look-ahead* heuristics, which rank variables on the basis of the effect of their assignment in the subsequent part of the computation.

Look-back heuristics are, in general, easier to implement. It suffices to gather, during the computation, some information about the assignments and their effects (e.g., the simplifications

of the input formula enabled by the performed assignments, or some statistics about failures, etc.). The overhead involved by these heuristics is small w.r.t. the whole computation. Conversely, look-ahead heuristics require a (partial) exploration of the "future" of the computation in order to determine the potential impact of alternative choices the solver can make at a choice point. This usually amounts to speculatively performing some steps of unit propagation. Albeit, in principle, look-ahead heuristics may lead to better choices (those that speed up the search the most), they also involve higher computational overhead, especially in a sequential implementation. This is a reason why look-ahead heuristics have largely been abandoned in modern solvers, in favor of the "lighter", but possibly coarser, look-back heuristics.

2.2. Look-ahead Heuristics

Look-ahead heuristics score variables depending on the effect their assignment has on the *current* state of the search. These heuristics can be considered as greedy algorithms: they evaluate, with respect to some *estimation function*, the alternative possible choices and select the most effective/promising one. If the best ranked option is not unique, one could select any of the best-ranked variables. In our implementation, we force determinism by selecting the variable with the lowest index. This way, the serial and the parallel implementations make the same choice, ensuring fairness in comparison.

Let us briefly recall the main families of look-ahead heuristics we are interested in.

Maximum Occurrences in Clauses of Minimum Size

These heuristics [11], briefly referred to as *MOM heuristics*, aim at selecting the unassigned variable that might impact the most in the subsequent unit propagation step, being present in "small" clauses. Variants of the schema appeared in the literature:

1. *Jeroslow-Wang heuristics (JW).* The goal of the JW heuristics is to select variable and value in such a way to maximize the chances of satisfying the formula [12]. This is made by computing the following weight function w for each literal ℓ:

$$\mathsf{w}(\ell) = \sum_{\omega \in \Phi \wedge\, \ell \in \omega} 2^{-|\omega|} \tag{1}$$

   where Φ is the current set of unresolved clauses and $|\omega|$ denotes the number of unassigned literals in the clause ω. There are two subvariants of JW:

   - *JW-OS (JW One-Sided)* considers the weights $\mathsf{w}(x)$ and $\mathsf{w}(\neg x)$ separately: the VSH selects the unassigned variable x having the largest individual weight (being it $\mathsf{w}(x)$ or $\mathsf{w}(\neg x)$).
   - *JW-TS (JW Two-Sided)* combines the weights $\mathsf{w}(x)$ and $\mathsf{w}(\neg x)$: the VSH selects the unassigned variable x having the largest $|\mathsf{w}(x) - \mathsf{w}(\neg x)|$ value.

   In both cases, the PSH assigns x true, if $\mathsf{w}(x) \geq \mathsf{w}(\neg x)$, false otherwise.
2. *BOHM heuristics* [13]. This heuristics associates to each unassigned variable x an array of weights $\langle \mathsf{w}_1(x), \mathsf{w}_2(x), \ldots, \mathsf{w}_n(x) \rangle$ such that, n is the number of literals in the largest

clause, and for each $i \in \{1, \ldots, n\}$:

$$\mathsf{w}_i(x) = \alpha \cdot \max\Big(lc_i(x), lc_i(\neg x)\Big) + \beta \cdot \min\Big(lc_i(x), lc_i(\neg x)\Big),$$

where $lc_i(\ell)$ denotes the number of occurrences of the literal ℓ in unresolved clauses of size i, and α and β are experimentally tuned parameters. The values used in [13] are $\alpha = 1$ and $\beta = 2$. The VSH selects the unassigned variable x having the maximum array (considering the lexicographical ordering). The PSH assigns x true if $\sum_i lc_i(x) \geq \sum_i lc_i(\neg x)$, false otherwise.

3. *PrOpositional SatIsfiability Testbed heuristics (POSIT)* [11]. This heuristics gives higher priority to unassigned variables having a high number of occurrences in the smallest unresolved clauses. This weight function is evaluated for each variable x:

$$\mathsf{w}(x) = lc_{min}(x) \cdot lc_{min}(\neg x) \cdot 2^{\eta} + lc_{min}(x) + lc_{min}(\neg x),$$

where $lc_{min}(\ell)$ denotes the number of occurrences of the literal ℓ in the smallest unresolved clauses, and η is a sufficiently large constant.[1] The VSH selects the unassigned variable x having the largest weight. The PSH assigns x false if $lc_{min}(x) \geq lc_{min}(\neg x)$, true otherwise. The aim of this heuristics is to maximize the effect of the unit propagation step that will follow the choice step.

Literal Count Heuristics

Briefly referred to as *LC heuristics* [14], their purpose is to select the variable whose assignment causes the satisfaction of the largest number of clauses. To this aim, they classify variables according to the number of their occurrences in unresolved clauses. Let $lc(\ell)$ denote the number of occurrences of ℓ in unresolved clauses. There are two main LC heuristics:

1. *Dynamic Largest Individual Sum (DLIS)* [15] that considers the values $lc(x)$ and $lc(\neg x)$ separately: the VSH selects the unassigned variable x having the largest value (being it $lc(x)$ or $lc(\neg x)$).
2. *Dynamic Largest Combined Sum (DLCS)* [14] that selects the unassigned variable x having the largest $lc(x) + lc(\neg x)$ value.

For both DLIS and DLCS, the PSH assigns x true if $lc(x) \geq lc(\neg x)$, false otherwise.

2.3. GPU and CUDA

The CUDA framework [3] is a general-purpose parallel computing platform and programming model that leverages the parallel computing engine of NVIDIA GPUs. It introduces a number of key abstractions that specifies, in particular,

- a hierarchical organization of threads (i.e., execution flows);

[1] According to [11], η should be such that 2^{η} is larger than the number of unresolved clauses, but small enough to avoid overflow in calculation of $\mathsf{w}(x)$. This allows the solver to enforce preference for variables x having a similar number of occurrences of x and $\neg x$.

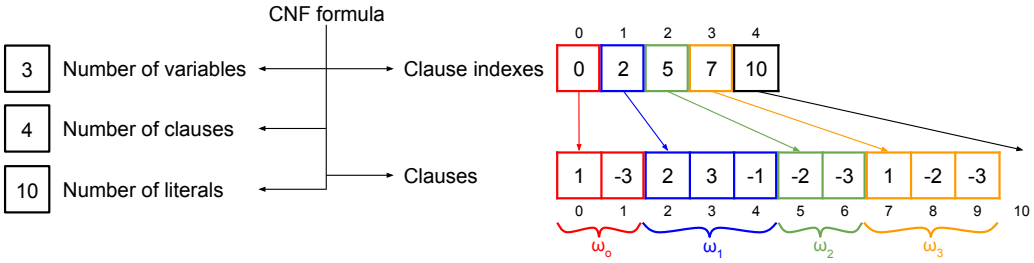

Figure 1: Data structure for the CNF formula $(x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$. Each variable x_i is represented by i, minus sign denotes negation.

- a hierarchy of memories (global, shared, constant, local, registers, etc.), with different scopes and lifetimes;
- some synchronization mechanisms.

These abstractions lead the programmer to partition the problem into subproblems that are independently solved in parallel by groups of threads, called *blocks*. In turn, the blocks are organized in *grids*. In particular, CUDA C extends the C language allowing the definition of particular functions, called *kernel*s. Kernels, identified by the use of the keyword `__global__` in their definition, are called by the host (CPU) and ran, in parallel, on multiple threads, on the device (GPU). The desired values of the size of grid and of the blocks per grid are passed as parameter:

```
kernelName<<<GridDim3D, BlockDim3D>>>(Actual Arguments).
```

The NVIDIA GPU architecture consists of thousands of identical compute units, called *cores*, grouped into a uniform collection of *Streaming Multiprocessors (SMs)*. SMs feature a *Single-Instruction Multiple-Thread (SIMT)* execution mode, designed to execute hundreds of threads concurrently. Each SM creates, schedules, and executes blocks of threads, further partitioned in groups of 32 threads, called *warps*. Threads in a warp are intended to execute in lock-step mode. However, each thread has its own program counter and register status. This allows each thread to branch out and execute independently, diverging from the execution of the other threads of its warp. The maximum performance is reached when thread divergence is avoided (so, all threads of a warp execute the same instruction) and when memory accesses patterns are designed to exploit the full bandwidth of each specific kind of memory. We refer the reader to [3] for a detailed presentation of CUDA and to [5, 6, 7] for descriptions of specific implementations of CUDA-based parallel solvers for SAT and ASP.

3. GPU-enhanced Look-ahead Heuristics

We have developed a CUDA C library, called *MiraCle*, implementing the look-ahead heuristics described in Section 2.2. For comparison purposes, the CPU versions have also been implemented using exactly the same data structures.

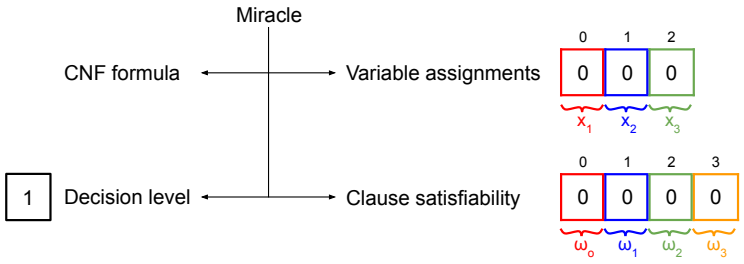

Figure 2: `Miracle` data structure `d_mrc` at (the starting) decision level 1. All variables are unassigned and all clauses are unresolved.

Data Structures Employed

A CNF formula (i.e., a set of clauses) is represented by the data structure depicted in Figure 1. It stores the numbers of variables, clauses, and literals, as well as a *clause array* and a *clause indexing array*. Each literal is represented by an integer index (negative values represent negative literals). Each clause is stored as the sequence of the indices representing its literals. Clauses are stored consecutively in the clause array. Their starting positions are recorded in the clause indexing array. Such a representation of a CNF formula is part of a larger data structure (called `Miracle` and instantiated as `d_mrc` in Figure 2 and in Table 1) which represents the state of the search in a specific moment in time. This data structure encompasses these parts:

- the CNF formula (as described in Figure 1);
- the current decision level (the starting decision level is 1);
- an array va storing the current variable assignments as signed integer values: for each variable i, $va[i] = \ell$ means that i has been assigned at level $|\ell|$ with polarity corresponding to the sign of ℓ. For unassigned variables we set $va[i] = 0$;
- the array cs storing information about clause satisfiability: $cs[j] = \ell$ if ω_j has been satisfied at decision level ℓ, or $cs[j] = 0$ if ω_j is unresolved.

MiraCle Implementation Details

The library provides functionalities that can be partitioned into three parts:

1. initialization and removal of the formula;
2. updating and restoring the formula;
3. computation of look-ahead heuristics.

We list the procedures of the three groups in Table 1. Due to space limitation, we cannot go into detail on all of them here. The interested reader can access the source code in http://clp.dimi.uniud.it/sw/, where they are documented. We restrict our presentation to those of the third group (see also the bottom part of Table 1). In particular, let us focus on the *JW One-Sided* heuristics, since the others essentially differ in the evaluation of the weighting function used to rank literals (cf., Section 2.2). This heuristics is computed by the function `mrc_gpu_JW_OS_heuristics()` (a simplified version of it is shown in Alg. 1 and illustrated in

`Miracle mrc_create_miracle(filename)`	Import a SAT instance in CNF format into the host data structure. Return a pointer to host memory.
`Miracle mrc_gpu_transfer_miracle_host_to_dev(mrc)`	Copy the data structure to the device. Return a pointer to device memory.
`mrc_destroy_miracle(mrc)` `mrc_gpu_destroy_miracle(d_mrc)`	Destroy host-side and device-side data structure, respectively.
`mrc_gpu_increase_decision_level(d_mrc)`	Increase the decision level (called before `mrc_gpu_assign_lits()`).
`mrc_gpu_assign_lits(lits, lits_len, d_mrc)`	Assign the literals `lits` true and update the data structure on the device.
`mrc_gpu_backjump(dl, d_mrc)`	Backjump to the decision level `dl` and update the device data structure.

`Lit mrc_gpu_JW_OS_heuristics(d_mrc)`
`Lit mrc_gpu_JW_TS_heuristics(d_mrc)`
`Lit mrc_gpu_BOHM_heuristics(d_mrc, a, b)`
`Lit mrc_gpu_POSIT_heuristics(d_mrc, n)`
`Lit mrc_gpu_DLIS_heuristics(d_mrc)`
`Lit mrc_gpu_DLCS_heuristics(d_mrc)`
	Compute heuristics on the device, w.r.t. the current assignment stored in `d_mrc` (`a,b,n` are the parameters α, β, η described in Section 2.2). Return the "best" literal.

Table 1
Main components of the *MiraCle* library (see Section 3).

Figure 3). The computation proceeds as follows. After resetting the working array `lit_weights` in global memory and configuring the launch parameters (w.r.t. the number of available SMs), the kernel `JW_weigh_lits_unres_clauses_krn()` is run. In its grid, each thread processes one or more clauses by adopting a grid-stride loop. For each unresolved clause, in lines 16–21, its contribute to the weight of each of its unassigned literals is computed according to (1). The global weights are updated using an atomic instruction for all unassigned literals (lines 22-24) to avoid race conditions. The best selectable literal and its weight are singled out by reducing the array `lit_weights`. The computation of the logarithmic reduction is performed on the GPU by `find_idx_max_float()`, called in line 7 (where, for readability, we improperly used a compact form to denote the retrieval of the result).

Integration Into a SAT Solver

The library has been designed by abstracting from CUDA implementation details and to be easily integrable into any DPLL or CDCL solver. Hence, its use does not require deep CUDA programming skills. Table 1 lists the basic functions that can be used to enhance a (serial) SAT solver with CUDA-based evaluation of look-ahead heuristics. To this aim, it suffices adding suitable calls into the code of the solver. More specifically, the solver has to first initialize the `Miracle` data structure by calling `mrc_create_miracle()`. This creates a representation of the SAT problem (cf., Figure 2) on host (called `mrc` in Table 1). Then, `mrc` is copied to device global memory, using `mrc_gpu_transfer_miracle_host_to_dev()`. This function returns a reference `d_mrc` to the device-side structure, that will be used by all subsequent calls to the library functions. Once `d_mrc` has been created, the solver can proceed as its original algorithm dictates, but each time the solver assigns a literal, increases the current decision level, and consequently performs some propagations, it has to update the device-side structure by calling `mrc_gpu_increase_decision_level()` and `mrc_gpu_assign_lits()` (the

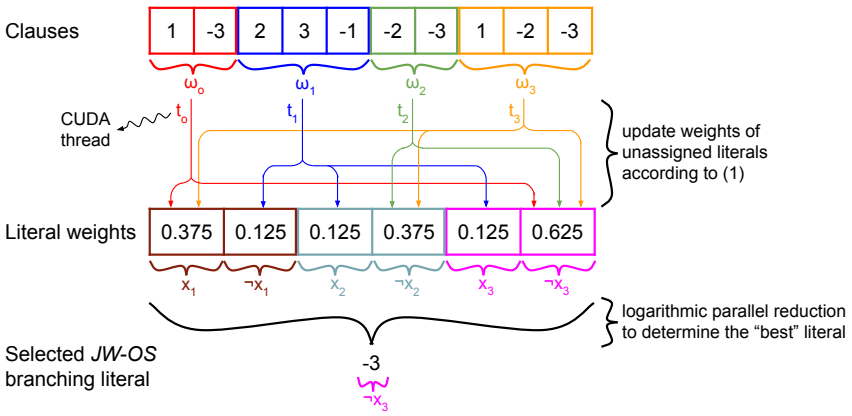

Figure 3: Computation scheme of the *JW One-Sided* heuristics on the GPU. First, each thread processes an unresolved clause and updates the weight of each unassigned literals in it according to (1). Then, the best selectable literal is computed by reducing the array of literal weights.

latter requires the list of assigned and propagated literals as argument). Similarly, whenever the solver performs a backjump to a level `dl`, the function `mrc_gpu_backjump()` should be used to update `d_mrc`. Each time the solver needs the GPU-based computation of one or more heuristics, the corresponding functions (listed in the bottom part of Table 1) can be called.

4. Experimental Results

To experiment with the library, we integrated it into two existing SAT solvers:

- *SATSolverDPLL*: a DPLL solver developed by Sukrut Rao [16]. This is a basic SAT solver implementing the raw DPLL algorithm. We have chosen this minimal implementation because its essentiality guarantees a fairer comparison between the various branching heuristics (in both their serial and parallel versions), not biased by the effect of other strategies, techniques, and optimizations that are often adopted in implementing a SAT solver.
- *microsat*: a CDCL solver originally developed by Marijn Heule and later modified by Armin Biere [17]. This is a simple conflict-driven SAT solver exploiting watched literals, clause learning, restart, and clause forgetting. It exposes greater performance w.r.t. the aforementioned *SATSolverDPLL* solver. We used this solver to compare the "quality" of the look-ahead heuristics against the lock-back heuristics used by *microsat*, namely VMTF (Variable Move-To-Front [18]).

To experiment with the two GPU-enhanced SAT solvers, we used a server equipped with an octa-core (16 threads) Intel i9-11900K 3.5GHz, with 16 MB cache and 64 GB DRAM, running Ubuntu 20.04.3 LTS (kernel 5.11.0). In this section, we report on experiments ran using a device NVIDIA GeForce RTX 3090 (compute capability 8.6, Ampere architecture, 24 GB, 82 SMs, 10496 CUDA-cores, clock rate 1.7 GHz). The code was compiled using GCC 9.3.0

Algorithm 1: Host and device code for evaluation of the JW-OS heuristics (simplified)

```
static Lit MRC_GPU_JW_OS_HEURISTICS(d_mrc){
    Data: nlits, num_clauses: number of clauses and literals
    Data: lit_weights: array for literal weights (device-side)
1   int blks, tpb;
    /* clear lit_weights:                                                    */
2   cudaMemset(lit_weights, 0, sizeof(float)*nlits);
    /* retrieve values blks and tpb, computed depending on GPU specs:        */
3   configureLaunchParam(num_clauses, &blks, &tpb);
    /* compute literal ranking on the device:                                */
4   JW_weigh_lits_unres_clauses_krn<<<blks, tpb>>>(d_mrc);
    /* logarithmic parallel reduction to determine the "best" literal:       */
5   Lidx blidx;                          /* Selected JW-OS branching literal index */
6   float lw_blidx;                                        /* and its weight */
    /* retrieval of the result:                                              */
7   (blidx,lw_blidx) = find_idx_max_float(lit_weights);
8   return ((lw_blidx == 0)?UNDEF_LIT:lidx_to_lit(blidx));
}

__global__ void JW_WEIGH_LITS_UNRES_CLAUSES_KRN(d_mrc){
    Data: ncls: number of clauses
9   float W;                                              /* weight of a literal */
10  Lidx lidx;                                    /* index of a literal in a clause */
11  register int c_size;              /* number of unassigned literals in a clause */
    /* pointers into d_mrc, kept in registers to speed up accesses:         */
12  register int * clss = d_mrc->clause_sat;
13  register int * vars = d_mrc->var_ass;
14  register int K; register int B;

15  for(i=threadIdx.x+blockIdx.x*blockDim.x; i<ncls; i+=blockDim.x*gridDim.x){
     /* using a stride-loop each thread processes one or more clauses       */
16     if(!(clss[i])){ /* if the clause is unresolved                        */
17         c_size = 0; B = cl_idxs[i]; K = cl_idxs[i+1];
18         for(int l = B; l < K; l++){ /* count unassigned lits of clause    */
19             lidx = cls[l];
20             if(!(vars[lidx_to_var(lidx)])) c_size++;
           }
21         W = exp2f((float)-c_size);                          /* compute weight */
22         for(int l = B; l < K; l++){ /* update weights of unassigned lits  */
23             lidx = cls[l];
24             if(!(vars[lidx_to_var(lidx)])) atomicAdd(&(lit_weights[lidx]),W);
           }
       }
    }
}
```

and CUDA 11.5. We also ran analogous experiments using other GPUs (such as, NVIDIA Tesla K40c and GeForce GTX 1060), obtaining results in line with those we present here.

The first experiment we report on tries to assess the "*quality*" of the heuristics, namely, how much the outcome of different heuristic selection functions affects the overall computation of SAT solvers. To this aim, we considered the number of heuristics calls a solver has to make before reaching a solution of a SAT instance. Intuitively, a lower number of calls suggests that the heuristics better drives the solver to the solution (or to the detection of unsatisfiability). To run this experiment, we used a dataset of instances from [19]. In particular, we considered 180 instances from the benchmarks *aim*, *Beijing*, *blocksworld*, *dubois*, *ii16*, *ii32*, *jnh*, *logistics*, *pigeon-hole*, *pret*, *ssa*. Figure 4 shows the comparison of the look-ahead heuristics and the native look-back

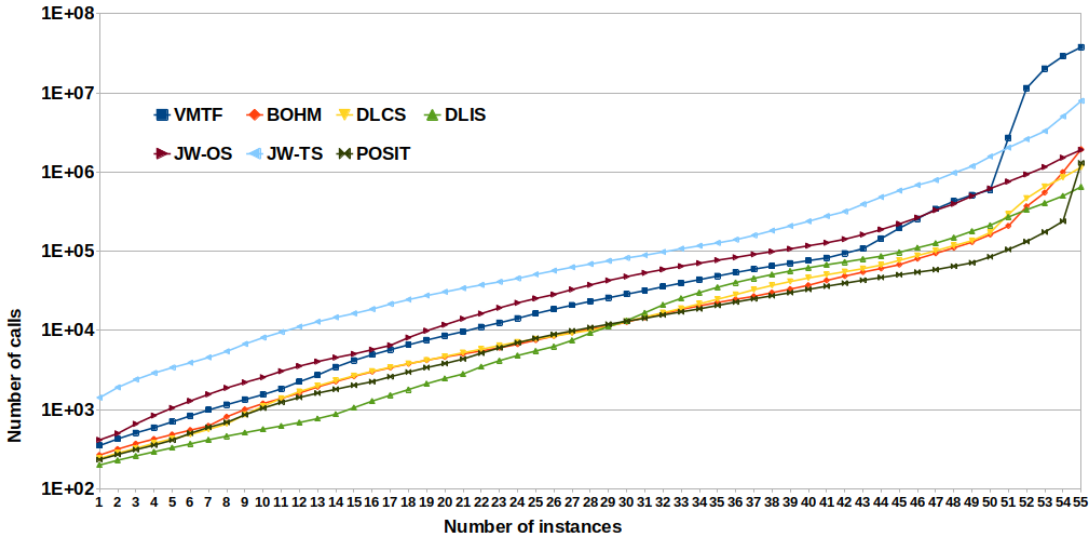

Figure 4: Number of calls to the heuristics needed by *microsat* to solve the instances, using the various look-ahead heuristics and its native look-back heuristics VMTF.

Id	Instance	Size (MB)	Variables	Clauses
I1	at-least-two-sokoban-sequential-p145-microban-sequential.030-NOTKNOWN.cnf	45	198252	2385409
I2	SC21_Timetable_C_557_E_73_Cl_37_S_35.cnf	63	406207	2841961
I3	Mycielski-11-hints-4.cnf	79	15350	3975330
I4	E00X23.cnf	104	15364	2133873
I5	spg_400_281.cnf	111	792025	4063559
I6	9dlx_vliw_at_b_iq8.cnf	161	371419	7170909
I7	vlsat2_702_14170.cnf	224	70288	14170788
I8	13pipe_k.cnf	239	147626	12295313
I9	crafted_n12_d6_c4_num17.cnf	246	56064	15834160
I10	blocks-blocks-36-0.180-SAT.cnf	247	733825	13169160
I11	sokoban-p16.sas.ex.19-sc2016.cnf	264	2929760	6312685
I12	barman-pfile10-040.sas.ex.15.cnf	405	430288	976816
I13	Kakuro-easy-115-ext.xml.hg_5.cnf	616	171688	24612456

Table 2
Excerpt of the set of instances used in performance comparison of CPU and GPU implementations of branching heuristics (see Figure 6).

heuristics of *microsat* (VMTF) for an excerpt of those instances whose computation finished within a timeout of 10 minutes, for each of the heuristics we used. The cactus-plot shows the cumulative number of calls (Y-axis) needed to solve a number of instances (X-axis). We observe that the worst performance are those of the two variants of Jeroslow-Wang heuristics, while all the others allow the solver to compute the solution using a significantly lower number of calls to the library functions (observe that the plot uses a log-scale for the Y-axis).

Similar results have been obtained for the solver *SATSolverDPLL*, as can be observed from the analogous cactus-plot shown in Figure 5.

To compare the performance of CPU and GPU implementations of the look-ahead heuristics in

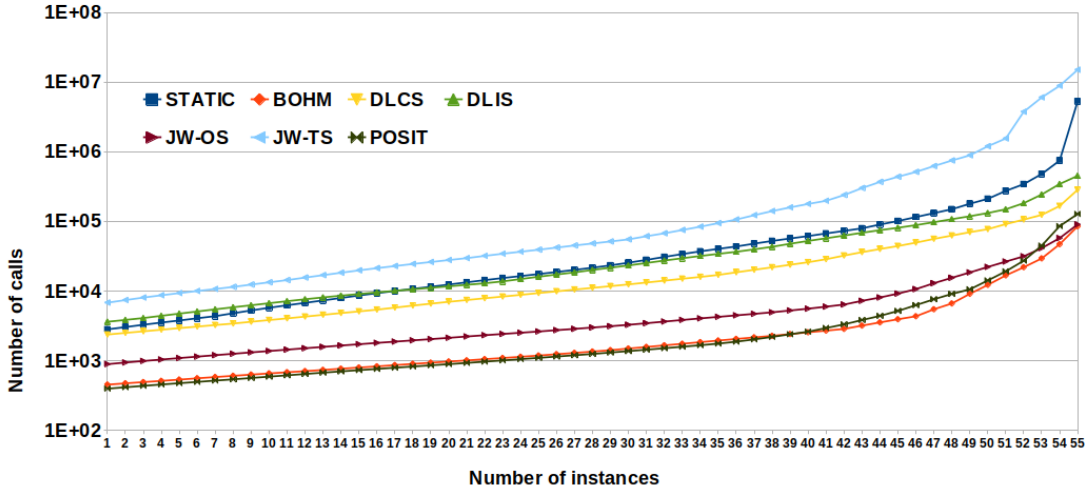

Figure 5: Number of calls to the heuristics needed by *SATSolverDPLL* to solve the instances, using the various look-ahead heuristics and its native heuristics (STATIC, in the chart). Such a native static branching heuristics is the same as DLIS but is executed once for all at the beginning of the computation.

MiraCle, we ran *microsat* on a selection of instances from [20] and evaluated the average time spent in computing each branching literal. We only considered instances having size greater than 40 MB. Figure 6 reports the comparison of the time spent by the 7 different implementations of the BOHM heuristics for a significant excerpt of the dataset (see Table 2). We observe that all parallel implementations outperform the serial one, obtaining 100x average speedup. The best performance are obtained by using 512 threads-per-block: 136x average speedup and 534x maximum speedup. We also notice that the number of threads-per-block significantly influences the performance of the solver in almost all instances. In fact, for each instance, Figure 6 shows how the average time decreases as the number of threads-per-block increases (note that the plot uses a log-scale for the Y-axis). Analogous results have been obtained for the other look-ahead heuristics (charts omitted because of space limits).

The results of the experiments seem to confirm that a GPU-based parallel implementations of a look-ahead heuristics can provide better performance in heuristic function evaluations. Moreover, as expected, the quality of the outcome of these branching heuristics is in general greater than those of the look-back options. Even when this is not the case, as for Jeroslow-Wang (cf., Figs. 4 and 5), the sub-optimal choices in literal selection is compensated by a higher efficiency in heuristics computation (cf., Figure 6).

5. Concluding Remarks

Due to the high computational cost of look-ahead heuristics, designers of modern SAT solvers tend to prefer alternative look-back heuristics. In an attempt to rehabilitate look-ahead heuristics, we described a GPU-based C library, *MiraCle*, implementing CUDA versions of the main

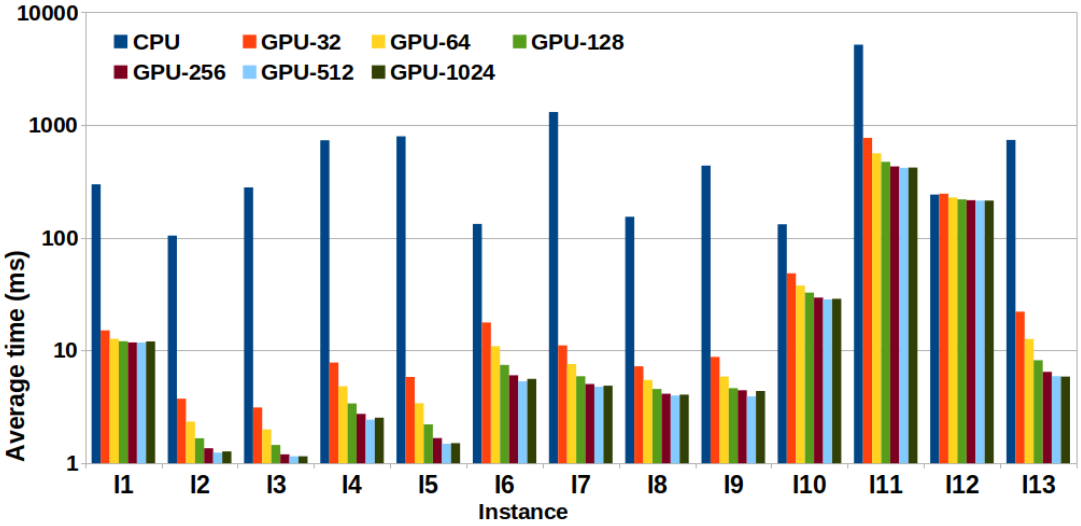

Figure 6: Average time spent in each BOHM heuristics evaluation by the CDCL solver. X-axis: an excerpt of the instances we experimented with. Y-axis: average time in ms. For each instance, we compare the time spent by the CPU implementation and by 6 versions of the CUDA implementation (using 32, 64, 128, 256, 512, and 1024 threads-per-block, resp.).

branching heuristics. We have shown the feasibility of the proposal by realizing the integration of the GPU-based functionalities into two different SAT solvers. Experimentation carried out on a significant number of instances has emphasized how CUDA implementations of the most common look-ahead heuristics can fully exploit the computational power of graphics cards. This allows to enhance a generic SAT solver (not necessarily parallel) by providing it with the parallel computation of such branching heuristics, that, in a purely serial context, would represent an inefficient computational bottleneck. Ameliorations of the library are currently under development. For instance, we plan to improve the implementation of the functionalities offered by *MiraCle* by introducing optimizations that depend on the compute capability of specific GPU in use (e.g., the possibility of exploiting cutting-edge technologies such as *tensor cores* and *warp-level* optimized intrinsic functions). We intend to extend the library by considering other heuristics that appeared in the literature, to investigate whether they might benefit from a parallel implementation. Some examples are the Clause Reduction Heuristics (CRH) [21] proposed in *OKsolver*, the Weighted Binaries Heuristics (WBH) [22] applied in the solver *Satz*, and the Backbone Search Heuristics (BSH) [23]. This would be a first step toward the realization of a purely GPU-based full-blown SAT solver, by means of an integration of *MiraCle* into the existing parallel solvers, such as, for example, the one described in [5].

Other lines of research can benefit from the experience made in designing and improving *MiraCle*. In fact, many of the software solutions designed and implemented to realize a (library supporting) GPU-based SAT solving, have application in the broader field of Computational Logic [24, 25]. We intend to adopt and adapt the approach we described in this paper for SAT, to the prototypical GPU-based solvers we proposed in past research for Answer Set Programming [26, 27] and Constraint Solving [28, 29, 30].

References

[1] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Communications of the ACM 5 (1962) 394–397.

[2] A. Biere, M. Heule, H. van Maaren, T. Walsh, Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[3] NVIDIA, CUDA C++: Programming Guide (v.11.6), NVIDIA Press, Santa Clara, CA, 2022.

[4] A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli, Exploiting unexploited computing resources for computational logics, in: F. A. Lisi (Ed.), Proc. of the 9th Italian Convention on Computational Logic, volume 857 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012, pp. 74–88. URL: http://ceur-ws.org/Vol-857/paper_f06.pdf.

[5] A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli, CUD@SAT: SAT solving on GPUs, J. of Experimental & Theoretical Artificial Intelligence (JETAI) 27 (2015) 293–316.

[6] A. Dovier, A. Formisano, E. Pontelli, Parallel answer set programming, in: Y. Hamadi, L. Sais (Eds.), Handbook of Parallel Constraint Reasoning, Springer, 2018, pp. 237–282.

[7] A. Dovier, A. Formisano, F. Vella, GPU-based parallelism for ASP-solving, in: P. Hofstedt, S. Abreu, U. John, H. Kuchen, D. Seipel (Eds.), Revised Selected Papers from DECLARE 2019, volume 12057 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 3–23.

[8] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, 1971, pp. 151–158.

[9] L. A. Levin, Universal sequential search problems, Problemy Peredachi Informatsii 9 (1973) 115–116.

[10] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7 (1960) 201–215.

[11] J. W. Freeman, Improvements to Propositional Satisfiability Search Algorithms, Ph.D. thesis, University of Pennsylvania, 1995.

[12] R. G. Jeroslow, J. Wang, Solving propositional satisfiability problems, Annals of Mathematics and Artificial Intelligence 1 (1990) 167–187.

[13] M. Buro, H. Kleine Büning, Report on a SAT competition, Bulletin of EATCS 49 (1992).

[14] J. P. Marques-Silva, The impact of branching heuristics in propositional satisfiability algorithms, in: Portuguese Conference on Artificial Intelligence, Springer, 1999, pp. 62–74.

[15] J. P. Marques-Silva, K. A. Sakallah, GRASP — a new search algorithm for satisfiability, in: The Best of ICCAD, Springer, 2003, pp. 73–89.

[16] S. Rao, SAT-Solver-DPLL, github.com/sukrutrao/SAT-Solver-DPLL. Last accessed in 2022.

[17] M. Heule, A. Biere, Microsat, github.com/arminbiere/microsat. Last accessed in 2022.

[18] L. Ryan, Efficient Algorithms for Clause-Learning SAT Solvers, Master's thesis, Simon Fraser University, 2004.

[19] H. Hoos, SATLIB - benchmark problems, www.cs.ubc.ca/~hoos/SATLIB/benchm.html, 2022.

[20] M. Heule, M. Järvisalo, M. Suda, M. Iser, T. Balyo, N. Froleyks, SAT Competition 2021, satcompetition.github.io/2021, 2021.

[21] O. Kullmann, Investigating the behaviour of a SAT solver on random formulas, Technical Report CSR 23-2002, University of Wales Swansea, Swansea Wales, UK, 2002

[22] C. M. Li, et al., Look-ahead versus look-back for satisfiability problems, in: International

Conference on Principles and Practice of Constraint Programming, Springer, 1997, pp. 341–355.

[23] O. Dubois, G. Dequen, A backbone-search heuristic for efficient solving of hard 3-SAT formulae, in: IJCAI, volume 1, 2001, pp. 248–253.

[24] G. Gupta, E. Pontelli, K. A. M. Ali, M. Carlsson, M. V. Hermenegildo, Parallel execution of Prolog programs: a survey, ACM Trans. Program. Lang. Syst. 23 (2001) 472–602.

[25] A. Dovier, A. Formisano, G. Gupta, M. V. Hermenegildo, E. Pontelli, R. Rocha, Parallel logic programming: A sequel, Theory and Practice of Logic Programming (2022) 1–69. doi:10.1017/s1471068422000059.

[26] A. Dovier, A. Formisano, E. Pontelli, F. Vella, A GPU implementation of the ASP computation, in: M. Gavanelli, J. H. Reppy (Eds.), PADL 2016, volume 9585 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 30–47.

[27] A. Dovier, A. Formisano, E. Pontelli, F. Vella, Parallel Execution of the ASP Computation, in: M. De Vos, T. Eiter, Y. Lierler, F. Toni (Eds.), Tech.Comm. of ICLP 2015, volume 1433, CEUR-WS.org, 2015.

[28] A. Dovier, A. Formisano, E. Pontelli, F. Tardivo, {CUDA}: Set constraints on GPUs, Rendiconti dell'Istituto di Matematica dell'Università di Trieste 24 (2021).

[29] F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto, E. Pontelli, Exploring the Use of GPUs in Constraint Solving, in: M. Flatt, H. Guo (Eds.), Proc. of PADL 2014, volume 8324 of *Lecture Notes in Computer Science*, Springer, San Diego, CA, USA, 2014, pp. 152–167. URL: http://dx.doi.org/10.1007/978-3-319-04132-2_11. doi:10.1007/978-3-319-04132-2_11.

[30] F. Tardivo, A. Dovier, A. Formisano, L. Michel, E. Pontelli, Constraints propagation on GPU: A case study for AllDifferent, in: R. Calegari, G. Ciatto, A. Omicini (Eds.), Proc. of the 37th Italian Conference on Computational Logic (CILC 2022), CEUR Workshop Proceedings, CEUR-WS.org, 2022.