CUD@SAT: SAT Solving on GPUs

Alessandro Dal Palù Agostino Dovier Andrea Formisano Enrico Pontelli

July 9, 2014

Abstract

The parallel computing power offered by *Graphical Processing Units (GPUs)* has been recently exploited to support general purpose applications—by exploiting the availability of general API and the SIMT-style parallelism present in several classes of problems (e.g., numerical simulations, matrix manipulations)—where relatively simple computations need to be applied to all items in large sets of data. This paper investigates the use of GPUs in parallelizing a class of search problems, where the combinatorial nature leads to large parallel tasks and relatively less natural symmetries. Specifically, the investigation focuses on the well-known *Satisfiability Testing (SAT)* problem and on the use of the NVIDIA CUDA architecture, one of the most popular platforms for *GPU computing*. The paper explores ways to identify strong sources of GPU-style parallelism from SAT solving. The paper describes experiments with different design choices and evaluates the results. The outcomes demonstrate the potential for this approach, leading to one order of magnitude of speedup using a simple NVIDIA platform.

1 Introduction

Most modern computing architectures are equipped with a multicore, programmable *Graphic Processing Unit (GPU)*. GPUs provide common desktops and laptops with a significant number of computing cores (e.g., 48–512). The computational power of GPUs is fully exploited by graphic-intensive applications—often by efficiently supporting standard graphical APIs—but GPUs are often idle during execution of traditional applications. Moreover, extra GPUs can be added at a very low price per processor. This has inspired an extensive line of research dealing with the applicability of the parallel capabilities of the GPUs to solve *generic* (i.e., not related to graphic rendering) problems [42, 40].

In this work, we explore the use of GPUs to enhance performance in the resolution of search-oriented problems. In particular, we focus on *Propositional Satisfiability Testing* (briefly, *SAT*) [16]. SAT is the problem of determining whether a propositional logical formula φ, typically in *conjunctive normal form*, is satisfiable—i.e., there is at least one assignment of Boolean values to the logical variables present in φ such that the formula evaluates to *true*.

Traditional approaches to parallel SAT explored in the literature rely on general purpose parallel architectures—i.e., either *tightly coupled* architectures (e.g., multicore or other forms of shared memory platforms [8, 34, 21]) or *loosely coupled* architectures (e.g., Beowulf clusters [46, 7, 14, 32, 23, 43]). The former type of architecture is commonly found in most desktops and servers. The availability of shared memory enables relatively simple parallelization schemes; e.g., it is easy to derive effective dynamic load balancing solutions. On the other hand, shared memory platforms tend to pose limits to scalability, due to the relatively small number of processors that can be linked to a central bus and/or a centralized memory bank. Loosely coupled platforms offer greater opportunities to scale to several hundreds or thousands of computing cores, but are more expensive, relatively less commonly available, and the communication costs require more expensive trade-offs in terms of load balancing. The use of GPUs offsets some of these problems: GPUs provide large numbers of processing elements, they are available in commodity architectures, and they provide a centralized view of memory.

The majority of parallel SAT solutions proposed in the literature are *task-parallel* solutions, where parts of the search space (i.e., sets of alternative variable assignments) are asynchronously explored by different processors [28, 44]. On the other hand, GPUs have been predominantly used for *data-parallel* computations—where parallelism relies on the concurrent application of a single operation/thread on different items drawn from a large data set. The mapping of a parallel exploration of the SAT search space on GPUs is a challenge we address in this paper. Furthermore, the architectural organization of GPUs (in terms of organization of cores, processors, and memory levels) is radically different, *making existing techniques proposed for parallel SAT inapplicable.*

The goal of this paper is to demonstrate that very fine-grained parallelism, which is unsuitable to traditional forms of parallelism investigated for SAT, can be effectively exploited by the peculiar architecture provided by GPUs. In this paper, we propose a first approach to this problem by designing a GPU-based version of the *Davis-Putnam-Logemann-Loveland (DPLL)* procedure [18]—DPLL is at the core of the majority of existing SAT solvers [6]. We show how both the unit-propagation stage and the search stage can be parallelized using GPUs. We also explore the use of learning techniques and variable selection heuristics, and prove the effectiveness of their interactions with GPUs computations. The proposed solution builds on the specific computational and memory model provided by NVIDIA's *Compute Unified Device Architecture (CUDA)* architecture [41], one of the most popular platforms for general purpose GPU-computing. We use different NVIDIA GPU platforms, ranging from highly specialized ones to those used in a typical office desktop, and we demonstrate that we easily reach speedups of one order of magnitude.

While we focus in this paper on the specifics of SAT, we would like to emphasize the similarities underlying SAT and other problems involving search, such as constraint-solving over finite domains [27, 1] and Answer Set Programming [2]. As a matter of fact, during search, these paradigms alternate non-deterministic assignment and polynomial-time propagation stages. In the long term, we expect to extend the experiences acquired working with SAT to these other domains. Therefore, we did not implement every optimization and strategy of a state-of-the-art SAT solver, while we dedicated to the most critical aspects that are of interest for parallel computation and to the capabilities that a GPU device can be employed.

The rest of the paper is organized as follows. Section 2 provides a brief overview of CUDA. Section 3 summarizes the DPLL algorithm. Section 4 describes our implementation of DPLL on GPUs. Section 5 provides an experimental evaluation to demonstrate the potential offered by GPUs in SAT solving. Section 6 discusses the relevant existing literature, while Section 7 provides conclusions and directions for future research.

2 CUDA: A Brief Overview

Modern graphic cards (*Graphics Processing Units*) are true multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy to support graphical processing (e.g., DirectX and OpenGL APIs). Efforts like NVIDIA's *CUDA—Compute Unified Device Architecture* [41, 33] aimed at enabling the use of the multicores of a graphic card to accelerate general (non-graphical) applications—by providing programming models and APIs that enable the full programmability of the GPU. This movement allowed programmers to gain access to the parallel processing capabilities of a GPU without the restrictions of graphical APIs. In this paper, we consider the CUDA programming model proposed by NVIDIA. The underlying conceptual model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT)*, a variant of the popular SIMD model; in SIMT, the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread. CUDA's architectural model is represented in Figure 1.

2.1 Hardware Architecture

A general purpose GPU is a parallel machine. 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 class of GPU—e.g., the Fermi architecture, shown in Figure 1 on the right, provides 16 SMs. In turn, each SM contains a collection of computing cores, typically containing a fully pipelined ALU and floating-point processing unit. The number of cores per SM may range from 8 (in the older G80 platforms) to 32 (e.g., in the Fermi platforms). Each GPU provides access to both on-chip memory (used for thread registers and shared memory—defined later) and off-chip memory (used for L2 cache, global memory and constant memory).

2.2 Host, Global, Device

The CUDA architecture recognizes two entities capable of performing computation—a core in a traditional CPU (referred to as the *Host*) and the cores of the GPU (referred to as the *Device*). As a result, a CUDA program is a high-level program that is decomposed in parts that can be run on the host and parts that can be executed on the device. Thus, the CUDA program is in charge of managing both sets of computing resources. From now on, we will focus on the typical structure of a C-based CUDA program.

Every function in a CUDA program is labeled as `host`, `global`, or `device`. A `host` function runs in the host and it is a standard C function. A `global` function is called by a `host` function but it runs on the device.

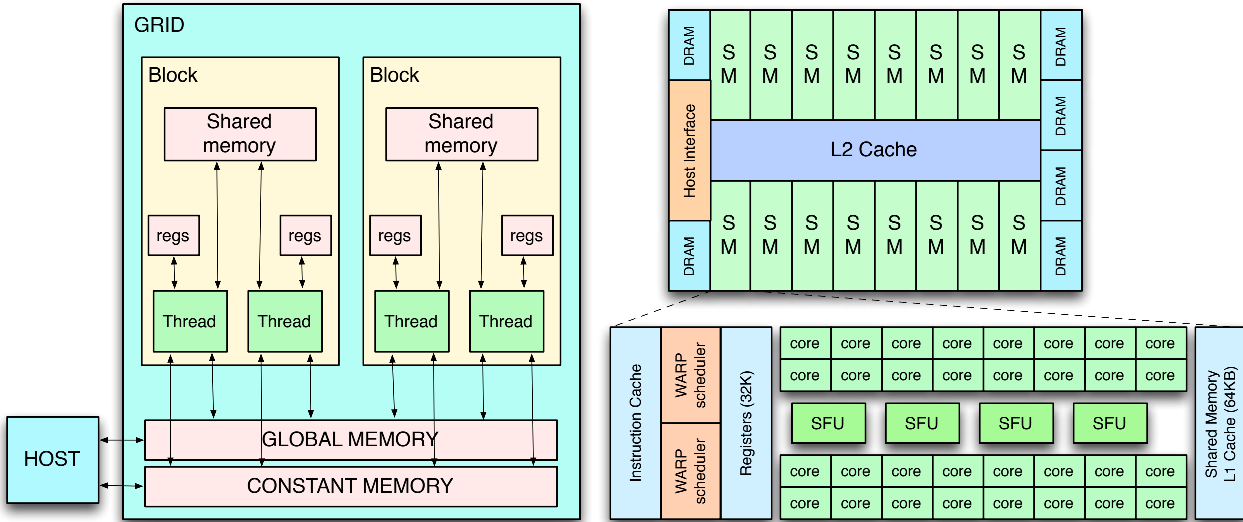

Figure 1: CUDA Logical Architecture (left) and Fermi Hardware Architecture (right)

A `global` function is also referred to as a *CUDA kernel function*. A `device` function can be called only by a function running on the device (`global` or `device`) and it runs on the device as well. Functions running on the device must adhere to some restrictions, depending on GPU's capabilities. In particular, limitations about the number of registers per thread and shared memories should be carefully handled.

2.3 Grid, Blocks, Threads

In order to create a uniform and independent view of the hardware, a logical representation of the computation is used: a *CUDA kernel function* describes the parallel logical tasks. When a kernel is called from the host program, a number of parameters are provided to indicate how many concurrent instances of the kernel function should be launched and how they are logically organized. The organization is hierarchical (see Figure 1 on left). The set of all these executions is called a *grid*. A grid is organized in *blocks* and every block is organized in a number of *threads*. The thread is, therefore, the basic parallel unit of execution. When mapping a kernel to a specific GPU, the original kernel blocks are scheduled on the multiprocessors of the GPU, and each group of threads in a block is partitioned and run as a *warp* (typically made of 32 threads), which is the smallest work unit on the device. Large tasks are typically fragmented over thousands of blocks and threads, while the GPU scheduler provides a lightweight switch among blocks.

The number of running blocks (gridDim) and the number of threads of each block (blockDim) is specified by the kernel call that is invoked on the host code with the following syntax:

$$\text{Kernel} \;\; \text{gridDim, blockDim} \; (\text{param}_1, \ldots, \text{param}_n);$$

These dimensions are described by the programmer. For instance, if the programmer states that `gridDim(8,4)` and `blockDim(2,4,8)`, then the grid contains in total $2,048$ threads. In this case, blocks are indexed by indices of the form (a, b), while each block is associated to threads indexed by triplets (i, j, k). When a thread starts, the variables `blockIdx.x` and `blockIdx.y` contain its unique block 2D identifiers, while `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` contain its unique thread 3D identifiers. We refer to these five variables as the *thread coordinates*. Thread coordinates are usually employed to enable the thread to identify its local task and its local portions of data. For example, many matrix operations, histograms, finite element simulations, and numerical simulations can be split into independent computations that are mapped to different logical blocks. From the efficiency standpoint, some caveats exist. In particular, since the computational model is SIMD (Single Instruction Multiple Data), it is important that each thread in a warp executes the same branch of execution. If this does not happen (e.g., two threads execute different branches of an `if-then-else` construct) the degree of concurrency will decrease. This phenomenon, called *branch divergence*, have been deeply investigated and various techniques to reduce its impact have been developed. The interested reader

can refer, for instance, to [26, 13, 19] and the references therein, for an analysis of the inefficiency originating from branch and thread divergence.

2.4 Memory Levels

Usually, host and device are distinct hardware units having physically distinct memories. These memories are connected by a system bus and DMA transfers can be used. When a host function calls a kernel function, the data required as input by the kernel computation has to be placed on the device. Depending on the type of GPU memory, there is an implicitly (through memory mapping) and/or explicitly (through the `cudaMemcpy` function) programmed transit of data between host and device memories.

The device memory architecture is rather involved, given the original purpose of the GPU—i.e., a graphical pipeline for image rendering. It features six different levels of memory, with very different properties in terms of location on chip, caching, read/write access, scope and lifetime: *(1)* registers, *(2)* local memory, *(3)* shared memory, *(4)* global memory, *(5)* constant memory, and *(6)* texture memory. Registers and local memory have a thread life span and visibility, while shared memory has a block scope (to facilitate thread cooperation); the other memory levels are permanent and visible from host and every thread. Constant and texture memories are the only memories to be read-only and to be cached.

The design of data structures for efficient memory access is the key to achieve good speedups, since access time and bandwidth of transfers are strongly affected by the type of memory and the sequence of access patterns (coalescing). Only registers and shared memory provide low latency, provided that shared accesses are either broadcasts (all threads read same location) or conflict-free accesses. The shared memory is divided into 16 different memory banks, that can be accessed in parallel. If two different threads request the same bank, then the requests are serialized, reducing the degree of concurrency. From the practical point of view, the compiler is in charge of mapping variables and arrays to the registers and the local memory. However, given the limited number of available registers per thread, an excessive number of variables will lead to variables being allocated in local memory, which is off-chip and significantly slower.

In the current GPUs, the constant memory is limited to 64KB and shared memory is limited to 48KB. In our initial tests, we used constant memory to store the input SAT formula; we quickly abandoned this approach, since reasonable size SAT input formulae easily exceed this amount of memory. Texture and global memories are the slowest and largest memories accessible by the device. Textures can be used in case data are accessed in an uncoalesced fashion—since these are cached. On the other hand, global accesses that are requested by threads in the same block and that cover an aligned window of 64 bytes are fetched at once.

In a complex scenario like this, the design of an efficient parallel protocol depends on the choice of the type of memory to use and the access patterns generated by the execution. For an interesting discussion of memory issues and data representations in a particular application (sparse matrix-vector product) the reader is referred to [5].

3 The Davis-Putnam-Logemann-Loveland (DPLL) Procedure

Let us briefly review the DPLL algorithm [18], here viewed as an algorithm to verify the satisfiability of a propositional logical formula in *conjunctive normal form (CNF)*.

Let us recall some preliminary definitions. An *atom* is a propositional variable (that can assume value `true` or `false`). A *literal* is an atom p or the negation of an atom ($\neg p$). A *clause* is a disjunction of literals $\ell_1 \vee \cdots \vee \ell_k$, while a CNF formula is a conjunction of clauses. The goal is to determine an assignment of Boolean values to (some of) the variables in a CNF formula Φ such that the formula evaluates to `true`.

Algorithm 1 provides a pseudo-code describing the DPLL algorithm. The two parameters represent the CNF formula Φ under consideration and the variables assignment θ computed so far—initially set as $\theta = []$, denoting the empty assignment. The procedure `unit_propagation` (Line 1) repeatedly looks for a clause in which all literals but one are instantiated to `false`. In this case, it selects the remaining undetermined literal ℓ, and extends the assignment in such a way to make ℓ (an thus the clause just being considered) `true`. This extended assignment can be propagated to the other clauses, possibly triggering further assignment expansions. The procedure `unit_propagation` returns the extended substitution θ'.

The procedure `ok` in Line 2 (respectively `ko` in Line 4) returns `true` if and only if for each clause of the (partially instantiated) formula Φ there is a true literal (resp., there is a clause whose literals are all false). The procedures `ok` and `ko` can be effectively computed during unit-propagation.

The procedure `select_variable` (Line 7) selects a un-instantiated variable using a heuristic strategy, in order to guide the search towards a solution.

Algorithm 1 `DPLL`(Φ, θ)

Require: Φ: CNF Formula
Require: θ: Substitution
1: $\theta' :=$ `unit_propagation`(Φ, θ)
2: **if** (`ok`$(\Phi\theta')$) **then**
3: **return** θ'
4: **else if** (`ko`$(\Phi\theta')$) **then**
5: **return** `false`
6: **else**
7: $X :=$ `select_variable`(Φ, θ')
8: $\theta'' :=$ `DPLL`$(\Phi, \theta'[X/$`true`$])$
9: **if** ($\theta'' \neq$ `false`) **then**
10: **return** θ''
11: **else**
12: **return** `DPLL`$(\Phi, \theta'[X/$`false`$])$
13: **end if**
14: **end if**

The recursive calls (Lines 8 and 12) implement the non-deterministic part of the procedure—known as the *splitting rule*. Given an unassigned variable X, the first non-deterministic branch is achieved by adding the assignment of `true` to X to θ'—this is denoted by $\theta'[X/$`true`$]$ (Line 8). If this does not lead to a solution, then the assignment of `false` to X will be attempted (Line 12). Trying first `true` and then `false`, as in Algorithm 1, is another heuristic decision.

Commonly used SAT solvers implement also learning capabilities—in the form of *clause learning* [3, 37]. After any (partial) assignment that falsifies one or more clauses is detected, the reasons for failure are analyzed and a minimal clause with the reason is added to the formula. This new clause prevents the same critical assignment to be explored again in the remainder of the search. This leads to a trade off between the (potentially exponential) space dedicated to the learned clauses and the search pruning induced by learning.

4 A GPU Implementation of a DPLL-based Solver

The SAT solver adopted in this study is based on a variant of the DPLL procedure described in the previous section, called `twolevel_DPLL` and summarized in Algorithm 2. The program runs part of the computation on the host and part on the device. The interaction between the two depends on an input parameter, called *execution mode* (*mode*). There are some possible execution modes:

- Mode 0: the computation *proceeds completely in the host*;
- Mode 1: each *unit-propagation* phase is executed, in parallel, on the GPU;
- Mode 2: the search in the lower part of the search tree is executed on the GPU (distributing subtrees of the search space to different threads);

Mode 3 combining modes 1 and 2 is also available, but it is not used in the tests since we aim at showing the potential of any of the two analyzed kinds of parallelism rather than their combination. Other parameters control learning capabilities and variable selection heuristics.

At the implementation level, the main data structures and the overall program structure of the CUDA program can be summarized as follows. We use the standard DIMACS format for the literal representation: a variable is represented by a positive integer number, while its Boolean negation by its arithmetical negation (0 is used as a clause separator by the DIMACS format). In the code we have developed, a formula is internally represented by:

- A vector called `formula`, that stores, sequentially, all the literals as they occur in the formula, represented by integer numbers as explained above.
- A vector called `clause_pointers`, that stores the pointers to the beginning of the clauses in `formula`;
- Three variables `NV`, `NL`, and `NC`, that store the number of variables, the number of literals, and the number of clauses in the formula, respectively.

Algorithm 2 `twolevel_DPLL`($\Phi, \theta, Mode$)

Require: Φ: CNF Formula
Require: θ: Substitution
Require: $Mode$: A parameter between 0 and 3

1: $\theta' :=$ `unit_propagation`($\Phi, \theta, Mode$)
2: **if** (`ok`($\Phi\theta'$)) **then**
3:    **return** θ'
4: **else if** (`ko`($\Phi\theta'$)) **then**
5:    **return** `false`
6: **else if** $((Mode > 1) \wedge (|$`FV`$(\Phi\theta')| <$ `MaxV`$))$ **then**
7:    **return** `CUDA_caller`(`filter_formula`($\Phi\theta'$))
8: **else**
9:    $X :=$ `select_variable`(Φ, θ')
10:    $\theta'' :=$ `twolevel_DPLL`($\Phi, \theta'[X/$`true`$], Mode$)
11:    **if** ($\theta'' \neq$ `false`) **then**
12:       **return** θ''
13:    **else**
14:       **return** `twolevel_DPLL`($\Phi, \theta'[X/$`false`$], Mode$)
15:    **end if**
16: **end if**

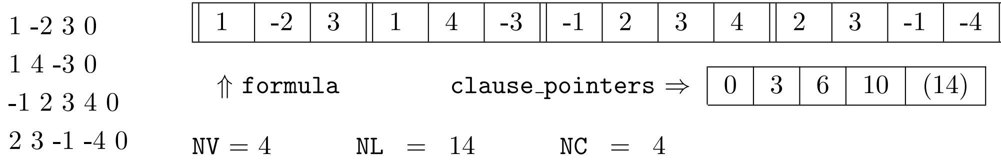

Figure 2: The simple SAT instance $(X_1 \vee \neg X_2 \vee X_3) \wedge \cdots \wedge (X_2 \vee X_3 \vee \neg X_1 \vee \neg X_4)$ represented using the DIMACS format (left) and by the internal data structure used in this paper (right)

An example is reported in Figure 2. A variable assignment is stored in an array of integers, called `vars`.

In the following subsections, we discuss in detail how parallelism is exploited from the unit-propagation (Section 4.1) and from the concurrent exploration of the tail of the search, i.e., the procedure `CUDA_caller` of Algorithm 2 (Section 4.2). The learning capabilities are briefly discussed in Section 4.3.

4.1 Unit-Propagation

The procedures `unit_propagation`, `ok`, `ko` of Algorithm 2 are implemented by a single function, called `mask_prop`. The unit-propagation is traditionally viewed as a fixpoint computation, which iterates the analysis of the clauses to detect either failures or to identify literals that can be deterministically assigned (i.e., clauses that have only one literal unassigned). The GPU implementation of unit-propagation is organized in two parts, one performed by the host and one delegated to the device. The host part is used to iterate the steps of the fixpoint computation, while the device part performs one step in the fixpoint computation.

Given a (partial) assignment θ stored in the `vars` array, function `mask_prop` computes an array of values, called `mask`, with an element for each clause of the original formula. For each i, the value of `mask`$[i]$ can be:

(a) 0, if the ith clause is satisfied by θ;

(b) -1, if all literals of the ith clause are falsified by θ;

(c) $u > 0$, if the ith clause is not yet satisfied by θ and there are still u unassigned literals in it.

Accordingly, the function `mask_prop` returns: -1 if there is a falsified clause; 0 if all of the clauses are satisfied; or a pointer to a literal in a clause i such that `mask`$[i] > 0$. The choice of this pointer, implementing the function `select_variable` (c.f., Algorithm 1–line 7 and Algorithm 2–line 9), can be computed by different heuristics (that can be selected within the `mask_prop` definition). Section 4.3 provides additional considerations concerning the implemented heuristics for this choice. In any case, clauses with `mask`$[i] = 1$ are always selected

first, allowing unit propagation: the function is called again till none of them are present (fixpoint of the unit propagation).

Since we are interested in a unique `mask`[i] result and a unique pointer, the `mask` array does not need to be built explicitly: a linear scan storing only the "best" information found is sufficient. Thus, a sequential implementation of the procedure runs in linear time in the number `NL` of (occurrences of) literals in the formula (namely, the size of the formula). Depending on the search parameters, this procedure is executed by the host (modes 0 and 2) or by the device (modes 1 and 3).[1]

In the remainder of the section, we briefly describe the choices we made in implementing a parallel version of the unit-propagation procedure. The size of the grid is based on a fixed number of blocks and a constant number of threads per block; this amount is limited by the GPU cores capabilities, e.g., 512. We explored other options, such as a unique block, or a number of blocks depending on instance size (e.g., `NC`/512), but we experimentally verified that, in such cases, the streaming multiprocessors (SM) were not exploited at best. In particular, we observed that the double of the number of available SMs is the optimal choice for the number of blocks (in our experimentation we choose 8 blocks).

The next item to be addressed is the identification of the structure of the tasks to be assigned to the different threads—which implies defining the amount of work that each thread will perform. The choice is guided by the two goals of *(a)* minimal thread interactions and *(b)* limited inter-block memory exchanges—performed either by the host or via the expensive global memory communication on the device. Several alternatives proved to be ineffective. The thread-to-literal mapping was soon discarded, because of the high degree of communication it requires. A mapping of a thread to a single clause requires a number of blocks that is proportional to the number of clauses in the formula; this may generate large queues for the SM schedulers, and will require additional effort by the host to collect and merge the data.

We eventually settled for a traditional *block partitioning,* where each block is assigned a segment of the formula, composed of a group of contiguous clauses. Each group contains 512 clauses that are assigned to the threads of a block. The partitioning of the formula follows a round-robin approach, whereby groups of clauses are cyclically assigned to the GPU blocks, until all groups have been assigned to a specific block.

Each thread is in charge of computing the `mask` value for a set of clauses and it stores (in local shared memory) the best literal to return, among those unassigned in the set of processed clauses. Note that different threads may process different numbers of literals, therefore the execution within a block might diverge. A synchronization barrier is enforced before applying a block *reduction*: a logarithmic scheme that sorts and extracts the best candidate among the results produced by the threads within the block. For instance, let us assume there are 16 threads (thread $0, \dots, 15$) in a block; at the first iteration, thread i with $i \in \{0, 2, 4, 6, 8, 10, 12, 14\}$ stores the best between the values computed by thread i and thread $i+1$, then thread i with $i \in \{0, 4, 8, 12\}$ stores the best between the values stored by thread i and thread $i+2$, then thread i with $i \in \{0, 8\}$ stores the best between the values stored by thread i and thread $i+4$, finally thread i with $i \in \{0\}$ stores the best between the values computed by itself and thread $i+8$, namely the best of all computed values. Therefore, after a number of iterations equal to the binary logarithm of the number of threads, a single thread in the block (thread 0 in the above example) is in charge to store the result in the GPU's global memory.

This mapping allows us to restrict the number of parallel reductions (one for block), as opposed to an approach where each thread is mapped to a clause, that would require a large number of blocks. Moreover, the chosen mapping limits the communication requirements, since the information transferred from the GPU back to the CPU is restricted to a constant number of elements (corresponding to the number of blocks). A final merge phase identifies the best result among all the blocks.

All the literals and clause pointers are transferred to the device's global memory at the beginning of the execution. Before the execution of this kernel function, the current variable assignment is copied to the device global memory, so that each thread can consult it. Since we also deal with learning, newly learned clauses need to be added incrementally during execution in both the host and the device (see Section 4.3). At the beginning of the execution of each block, the `clause_pointers` for each clause are read from the global memory in a coalesced fashion, which provides a significant speedup. However, this is not guaranteed for the clauses successively processed by the threads in the block, due to the divergence of the code (because of different choices and number of literals in the first clause).

Predicting the potential speedup of this approach is not easy. If we use k blocks, each block deals with `NC`/k clauses. Each block executes 512 threads. Recall that the number of blocks chosen is double than the number of SM and each SM has 48 computing cores. One has to consider CUDA's peculiar memory access, as well as the fact that the scheduler reasons at the warp level (groups of 32 threads). Moreover, each block has to wait

[1] We also investigated an alternative approach based on *watched literals*—a popular and successful technique used in sequential SAT solvers [6]. We will comment on the use of this technique in the GPU-based solver in Section 5.5.

```
addr := blockIdx.x; point := -1;
block_vars[0] := 0; count := 0;
for(i=1;i<NV;i++)
      if (count < B)
            { block_vars[i] = addr % 2;
              addr = addr/2; count++; }
      else { block_vars[i] = point;
              point--; }
```

Figure 3: Deterministic assignments of variables in the shared memory

for the thread that processes the largest number of literals (although we allocate them as much balanced as we can). The ideal speedup is bounded by the number of streaming multiprocessors multiplied by the size of a warp (i.e., $32 \times SM$).

For the sake of completeness, we mention that in the case of modes 2 and 3, the DPLL procedure is executed by each thread (see Section 4.2) and a specific sequential version of unit-propagation has been designed to be run by each thread. This version runs in time linear in the size of the input formula.

4.2 Parallelizing the Tail of the Search

Let us explore the effects of choosing the modes 2 and 3. The scenario occurs when, after the completion of a unit-propagation phase, the formula is neither satisfied or invalidated by the current partial variable assignment θ. If the number of unassigned variables is high, then the search continues on the host—by making a recursive call to `twolevel_DPLL`. If the number of unassigned variables is below a threshold `MaxV`, the remaining part of the search (the *tail* of the search) is delegated to a specific kernel function. To reduce the amount of data to be dealt with, the formula is first *filtered* using θ (call to `filter_formula` in Algorithm 2):

- If a clause is satisfied by θ, it is deleted from the formula;
- In all other clauses, if a literal is set to `false` by θ, then the literal is removed.

We choose two parameters `B` and `T` and executed the kernel code on $2^{\texttt{B}}$ blocks with $2^{\texttt{T}}$ threads per block. The meaning of `B` and `T` is explained below. In case a solution is found, the entire assignment is copied back to `vars`.

The first step of the parallel search consists of placing each block of threads in a different position of the search tree corresponding to the filtered formula. A shared vector `block_vars` is initialized using the binary expansion of the block coordinates, as illustrated in Figure 3. The first `B` variables are deterministically assigned by this process. Moreover, a unit-propagation phase (not shown in Figure 3, for the sake of simplicity) is performed after such assignment, in order to deterministically propagate the effect of the `block_vars` assignment.

After this initial assignment of values has been completed, the search task is delegated to each thread, through the execution of an iterative implementation of DPLL. Each thread in a block accesses the (fast) array `block_vars` and uses, for the other variables, the local (slower) array `delta_vars`. The variable `point` (see Figure 3) is used to take care of indirect references between these two data structures. During the iterative search, the first `T` choices are assigned deterministically by using the thread coordinates within a block (the value of `threadIdx.x`). Let us observe that, at this point, the various threads run in an independent manner, therefore in this case we do not take advantage of symmetric instructions and/or coalesced memory accesses. Experiments show that the computing power of the GPU for divergent branches can still provide a significant benefit to the search.

Let us conclude this section with a brief discussion of how the iterative device version of the DPLL procedure (and the backtracking process) is implemented. Each thread has a stack, whose maximum size is bounded by $\texttt{MaxV} - \texttt{B} - \texttt{T}$, where `MaxV` is the maximum number of variables sent uninstantiated to the device. This stack size corresponds to the maximum number of variables that need to be assigned during the execution of each thread in a block. Each stack element contains two fields: a pointer to `delta_vars` and a value. The value can be a number from 0 to 3, where 0 and 1 mean that the choice is no (longer) backtrackable, while 2 and 3 mean that a value (either 0 or 1) has been already attempted and the other value (1 or 0, respectively) has not been tried yet. Notice that this data structure allows us to implement different search heuristics. In case one adopts the simple strategy that always selects the value 0 first, the second field of the stack elements can be omitted.

4.3 Learning and Heuristics

Conflict-driven learning has been shown to be an effective method to speed up the search of solutions in SAT [6]. Whenever a value is assigned to the last unassigned variable x of a clause (w.r.t. the current partial assignment), a failure may arise (i.e., the clause might be falsified). In this case, x is said to be a *conflict variable*. As soon as a failure is generated, an explanation for such failure can be detected in a graph, called the *implication graph* [37]. The nodes of this graph represent variables, while the edges represent the clauses that "propagated" variable assignments. Starting from a conflicting clause, a sub-graph that traces the reasons for the assignments for its variables is built.

Because of the non-deterministic nature of the computation, some variables (called *decision variables*) have been explicitly assigned (through the `select_variable` procedure mentioned earlier), while other variables have been assigned by unit-propagation. Each decision variable is associated to a *decision level*, that reflects at what depth in the search tree the decision for such variable has been made (in other words, how many nested calls of the DPLL procedure have been performed before selecting such variable). The analysis of the sub-graph identifies a core of variables sufficient to falsify the conflicting clause. These core variables are used to derive a new clause, through suitable learning strategies, that, once added to the problem at hand, prevent visiting again the same subtree of the search space.

Among the possible learning strategies presented in the literature, we implemented the strategy *1-UIP,* originally adopted in GRASP [37]. Intuitively, this strategy identifies a *cut* in the implication graph that separates its *conflict side* (where the conflict variables/nodes occur) from its *reason side* (where the relevant decision variables occur). A *Unique Implication Point (UIP)* is a variable in the graph at the same decision level of the current assignment and such that every path from the decision variables of the level to the conflicting variable passes through it. The 1-UIP is the UIP closest to the conflict variable.

In terms of heuristics, we implemented the following variable selection heuristics, for both the host and the device code. Let us assume that all possible unit-propagations have been performed. Among the clauses with the minimum number of unassigned literals, we select the one in which a variable with minimum index occurs. If several clauses include the same minimal index variable, we break the tie by selecting the clause with the smallest index. The heuristics first tries the assignment that makes `true` the literal based on such variable. The host code also includes other well-known heuristics, such as the Jeroslaw-Wang [30] and its variants (see `http://clp.dimi.uniud.it/sw/cudasat/` for details).

5 Experiments

We report the results collected from some experiments using a preliminary prototype. The experiments are performed using three different hardware platforms:

- U: ("U"dine) *Host*: AMD Opteron 270, 2.01GHz, RAM 4GB, *Device*: NVIDIA GeForce GTS 450, 192 cores (4MP). Processor Clock 1.566GHz. Linux.
- F: ("F"ermi) *Host*: (12 core) Xeon e55645 at 2.4GHZ, 32GB RAM, *Device*: NVIDIA Tesla C2075, 448 cores (14MP). Processor Clock 1.15GHz. Linux.
- A: ("A" common desktop) *Host:* Intel i7, 64 bits, 3.4–3.7GHz, *Device:* NVIDIA GeForce GTX 560, 336 cores (7MP). Proc. Clock 1.62–1.9GHz. Windows 7.

In our experiments we do not exploit concurrency at the host level. The experiments investigate the impact of using CUDA for *(1)* parallelizing unit-propagation and *(2)* parallelizing the tail of the search. We report also *(3)* the behavior of parallelizing unit-propagation with respect to learning. We have also performed experiments where all the activities are delegated to the device. The programs and benchmarks are available at `http://clp.dimi.uniud.it/sw/cudasat/`.

5.1 Parallelizing Unit-Propagation

The parallelization of unit-propagation allows us to speed up the code when the formula becomes sufficiently large. In order to stress-test this feature (separately from the strategies used in the search), we need a small search tree with a large set of clauses. This setup can be achieved by replicating a set of clauses, so that the unit-propagations produce the same result, but proportionally to the size of the larger formula. This test supports the intuition that a GPU architecture has a communication overhead that needs to be amortized by an adequate amount of parallel work.

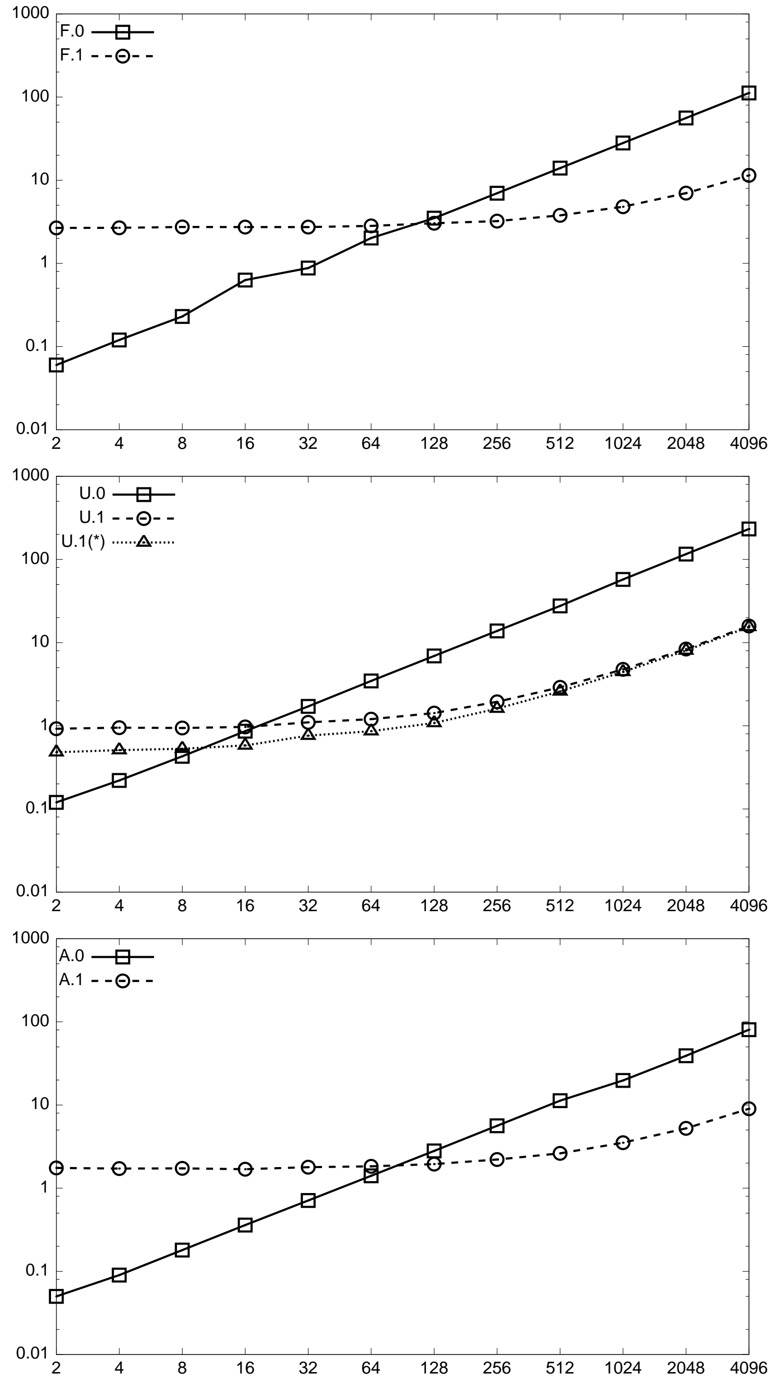

Figure 4: Unit-propagation tests. From top to bottom, architectures F, U and A are used respectively. Behavior of the running modes 0 (only host) and 1 (device unit-propagation). x-axis: size of the formula (1 is the size of `hole6`). y-axis: running time expressed in seconds.

Table 1: Complete table related to Figure 4

HW	2	4	8	16	32	64	128	256	512	1024	2048	4096
F.0	0.06	0.12	0.23	0.63	0.88	2.02	3.50	6.98	13.97	28.01	55.99	111.96
F.1	2.67	2.68	2.74	2.73	2.73	2.82	3.04	3.22	3.78	4.80	6.99	11.44
U.0	0.12	0.22	0.43	0.86	1.71	3.46	6.91	13.81	27.59	57.61	116	232.42
U.1	0.92	0.95	0.94	0.97	1.10	1.20	1.42	1.94	2.90	4.77	8.37	15.82
U.1(*)	0.48	0.51	0.53	0.58	0.76	0.86	1.08	1.61	2.59	4.45	8.03	15.47
A.0	0.05	0.09	0.18	0.36	0.71	1.41	2.81	5.63	11.27	19.77	39.16	80.49
A.1	1.75	1.72	1.73	1.69	1.79	1.83	1.95	2.21	2.62	3.52	5.24	9.02

For instance, we have chosen the pigeon-hole problem `hole6.cnf` (133 clauses, 42 vars, 294 literals) from SATLIB[2] and created files by replicating several copies of the original problem. We tested our code on such instances using the modes 0 (only host) and 1 (host, with unit-propagation parallelized on the device). A subset of the results is depicted in Figure 4 (see also Table 1). The figure describes the results for the unsatisfiable instance `hole6`. The different data sets are labeled with $X.Y$ where X is the hardware platform (F, U, A) and Y is the execution mode. The x axis indicates the number copies of the problem used (ranging from 2 to 4,096). The data set marked with (*) represents an execution completely performed on the device—further discussed in Section 5.4. Execution times (in seconds) are plotted on the y axis, using a logarithmic scale. The comparison focuses on the executions in mode 0 (which represent purely sequential executions of unit-propagation on the host) and executions in mode 1 (which parallelize the unit-propagation on the device).

Even if we noticed slightly different behaviors and speedups on the different platforms used, there is a common trend. We registered a speedup of 10 on the F machine, 15 on the U machine, and 9 on the A machine. We also experimented this option on the large instance set described in Section 5.2, where we found an average speedup of roughly 4 on medium size instances.

It can be noted in the graphs that for different configurations, similar trends are highlighted. However, for the GPU version, all performances include an initial overhead given by host-device interactions that is payed off only when the problem gets larger. Moreover, the problem size where the device computation (mode 1) gets more efficient than the host (mode 0) varies among different architectures. This fact depends on both CPU and GPU capabilities. For example, in the configuration F, notwithstanding the larger number of MP cores, there is a larger computation overhead for mode 1, when compared to other configurations. This is due to the slower GPU MP clocks and by the CPU in use.

Further experimentation involving parallel unit-propagation has been performed and reported in Table 5.3. We will comment on them in Section 5.3, where we investigate different options in performing unit-propagation and learning on the host and device.

5.2 Parallelizing the search

The behavior of CUDA during parallel search (mode 2) strongly depends on the values of the parameters `B`, `T`, and `MaxV`.

We recall that when the tail of the search is started, `B+T` variables' assignments are explored by means of 2^{B+T} parallel threads. Due to GPU's architectural limitations, the number of threads in a block is limited and hardware dependent (at most 1024). Therefore, we organized and studied flexible configurations that allow to distribute the work among blocks (`B` variables). In principle, there are no restrictions on the number of blocks, however it is convenient to have many threads per block, since they can exploit shared memory and convergent instructions features. In our tests, we show different combinations of `B` and `T`, to investigate the behavior and performances of the GPU.

Let us consider, for example, the unsatisfiable instance `x1_24.shuffled07` taken from the 2002 SAT Competition. In Figure 5 (see also Table 2), we report the running times obtained in detecting unsatisfiability and using different search settings on the U hardware platform. The x axis indicates different choices of `B` and `T`. The different curves correspond to different values for the parameter `MaxV`. The y axis records the execution time in seconds.

The running time for the computation performed completely by the host CPU for this problem is 1,116 seconds. The number of calls are 512, 256, 32, 8, 1 respectively for `MaxV` equal to 50, 55, 60, 65, 70. The best result in the chart is obtained with parameters (8, 8, 65), leading to a speedup of 38 over the sequential

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

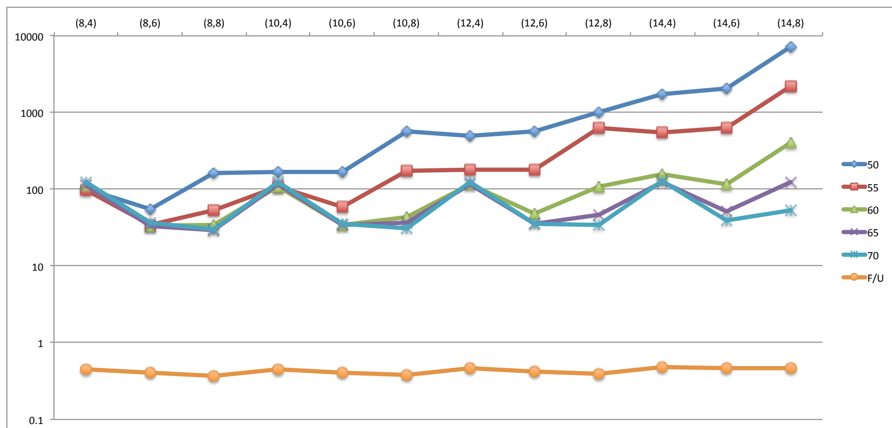

Figure 5: Execution times for different values of parameters (B, T) (x-axis), and MaxV on the x1_24.shuffled07 problem. The time (y coordinate) is in seconds.

Table 2: Complete table related to Figure 5

MaxV	GPU calls	(8,4)	(8,6)	(8,8)	(10,4)	(10,6)	(10,8)
50	512	102.54	54.65	161.18	166.44	167.17	577.81
55	256	97.32	34.45	53.63	109.00	58.41	174.11
60	32	110.53	33.66	34.42	108.95	34.44	42.89
65	8	113.91	33.47	**29.36**	116.97	34.90	36.94
70	1	122.63	36.20	29.69	122.96	35.42	30.66
F/U ratio		0.45	0.41	0.37	0.45	0.41	0.38

MaxV	GPU calls	(12,4)	(12,6)	(12,8)	(14,4)	(14,6)	(14,8)
50	512	504.16	576.32	1028.95	1768.54	2027.43	7215.4
55	256	180.41	181.74	630.57	557.63	635.52	2234.68
60	32	115.64	47.72	110.26	158.04	117.42	408.53
65	8	115.40	36.11	46.71	124.02	51.88	122.39
70	1	122.22	35.91	34.81	126.55	39.79	53.16
F/U ratio		0.47	0.42	0.40	0.48	0.47	0.46

execution time. The results are in line with those obtained by using the machines F and A—the bottom curve illustrates the ratio between the execution time obtained with the hardware F and that obtained on U. Let us observe that this ratio does not depend on the value of MaxV, while it is affected by the selection of the values for B and T.

In particular, as one may expect, delegating a larger portion of the search space to the device better exploits the most powerful hardware (namely, F in this case). More in general, one can observe that better speedups can be obtained by sending large tasks to the device—even if this is not a linear trend. This is rather counter-intuitive with respect to the standard guidelines for CUDA applications. We believe that the main reason is that our application deals with a limited amount of data, compared to the computational effort which is involved. Therefore, the simple usage of the cores as a parallel machine for divergent threads is still advantageous. On the contrary, for small values of MaxV, the high number of GPU calls (and consequent data transfer) causes a slowdown of the overall process.

We extensively experimented with several hundreds of instances and different configurations of the parameters B, T, and MaxV. The instances have been taken from the repository SATLIB and from the collections of instances used in the SAT competitions (specifically, the SAT-Competition 2011[3] and the SAT-Challenge 2012[4]).

[3] http://www.satcompetition.org/2011

[4] http://baldur.iti.kit.edu/SAT-Challenge-2012

Excerpt of the experimental results for satisfiable instances (timings are in seconds)

Instance	mode 0	variables	clauses	speedup	MaxV-B-T
BMS_k3_n100_m429_13 (SATLIB)	20.05	100	302	**3.2**	95-6-8
BMS_k3_n100_m429_37 (SATLIB)	10.70	100	303	**2.0**	95-8-8
RTI_k3_n100_m429_140 (SATLIB)	11.08	100	429	**4.6**	95-8-6
RTI_k3_n100_m429_483 (SATLIB)	10.65	100	429	**4.3**	100-7-7

Excerpt of the experimental results for unsatisfiable instances (timings are in seconds)

Instance	mode 0	vars	clauses	speedup	MaxV-B-T
marg3x3add8.shuffled-as.sat03-1449 (SAT-challenge12)	1242.68	41	224	**88.3**	35-6-7
marg3x3add8ch.shuffled-as.sat03-1448 (SAT03)	1751.00	41	272	**92.7**	35-6-7
battleship-5-8-unsat (SAT11)	2.26	40	105	**7.1**	40-8-8
battleship-6-9-unsat (SAT-challenge12)	69.38	54	171	**6.2**	55-10-8
unif-k5-r21.3-v50-c1065-S1449708927-022 (SAT11)	220.92	50	1065	**12.9**	50-6-7
unif-k5-r21.3-v50-c1065-S370067727-038 (SAT11)	213.95	50	1065	**11.4**	50-7-7
sgen1-unsat-61-100 (SAT-challenge12)	440.49	61	132	**7.8**	65-7-7
jnh16 (SATLIB)	53.14	100	850	**4**	100-8-8

The analysis of the results confirms what described earlier in the case of the specific instance x1_24.shuffled07. Tables 5.2 and 5.2 show a representative excerpt of the results concerning satisfiable and unsatisfiable instances, respectively. The triplet MaxV-B-T reported is the one corresponding to the fastest computation.

On average, a better speedup has been achieved in correspondence of unsatisfiable instances. This can be explained by considering that, in general, the satisfiable instances used in competitions are larger than the unsatisfiable ones. Hence, larger amounts of data have to be handled by each thread, leading to a higher degree of divergence among threads executions. Nevertheless, the speedups are remarkable for GPU-style executions.

Figure 6 depicts, for a set of four experiments, the relationships among the B, T, and MaxV parameters. As mentioned earlier, delegating larger portions of the problem to the device—i.e., when MaxV approaches the number of variables in the problem—seems to guarantee a better performance. With regards to the preferable values for the parameters B and T, the results emphasize two issues. First, the choice of a pair of values that are close to each other (e.g., $\mathtt{B} = \mathtt{T} \pm 1$), such that $6 \leq \mathtt{B} \leq 8$, very often guarantees greater speedups with respect to the complete execution on the host. Notice that this kind of choice for B and T generates device computations where the global number of threads is between 2^{11} and 2^{15}. Experiments where the computation uses either a smaller or a larger number of threads, revealed, in general, poorer performance. This last effect clearly appears to be connected to the specific hardware used (here, the machine U). It is reasonable to believe that running a larger number of threads might be appropriate when a more powerful device is used. In our case, as the value of $\mathtt{B} + \mathtt{T}$ grows beyond 15, we experience an increase in the number of failed computations (i.e., computations where a timeout is reached or device errors are generated because of excessive requests in device resources allocation).

A second issue we observe appears to be independent from the specific device in use: the use of values of T smaller than 5 does not translate to acceptable performance. This can be explained by noting that, since warps are composed of 2^5 threads, running blocks containing less than 32 threads does not allow a full exploitation of the parallelism offered by the device.

Tables 5.2 and 5.2 and data underlying Figure 6, are reported in greater details at http://clp.dimi.uniud.it/sw/cudasat/. In particular, the data in the site shows the experiments that produce the best speedups for each choice of MaxV. We also emphasize (highlighting numbers), for each instance, the best performance and the maximum speedup obtained (in the rightmost column). Figure 7 summarizes the results of Tables 5.2 and 5.2: for each instance and (B,T) values, we count the number of computations terminating in at most twice the duration of the fastest computation. These are the "good" computations and are represented by light gray areas. The number of times a parameters' configuration achieves the best result is shown by dark gray areas.

5.3 Parallelism in Unit-Propagation and Learning

Learning (and back-jumping) is a very effective way to enhance search. Since learning increases the size of the formula, CUDA parallelism applied to unit-propagation could become an interesting option. We have experimented with various instances from SATLIB. Some of the results are reported in Table 5.3. These experiments confirm this conjecture, since the GPU is capable of processing the high number of learned clauses with limited performance degradation. In particular, we can observe that the cost of sending the learned

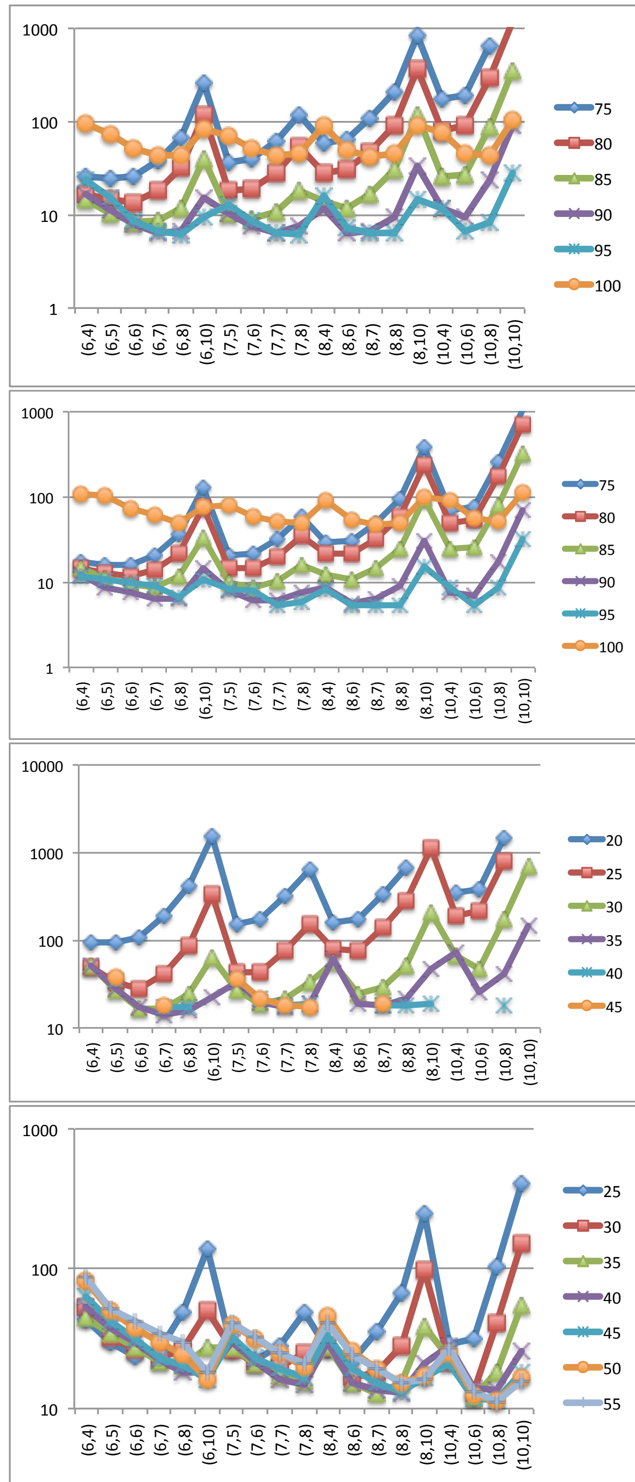

Figure 6: Some computational results for different values of the parameters (B,T) (x-axis), and MaxV for instances BMS_k3_n100_m429_13, BMS_k3_n100_m429_37 (from the top the first two, satisfiable), marg3x3add8.shuffled-as.sat03-1449 and battleship-6-9-unsat (the two on the bottom, unsatisfiable). Running times (y-axis) are in seconds.

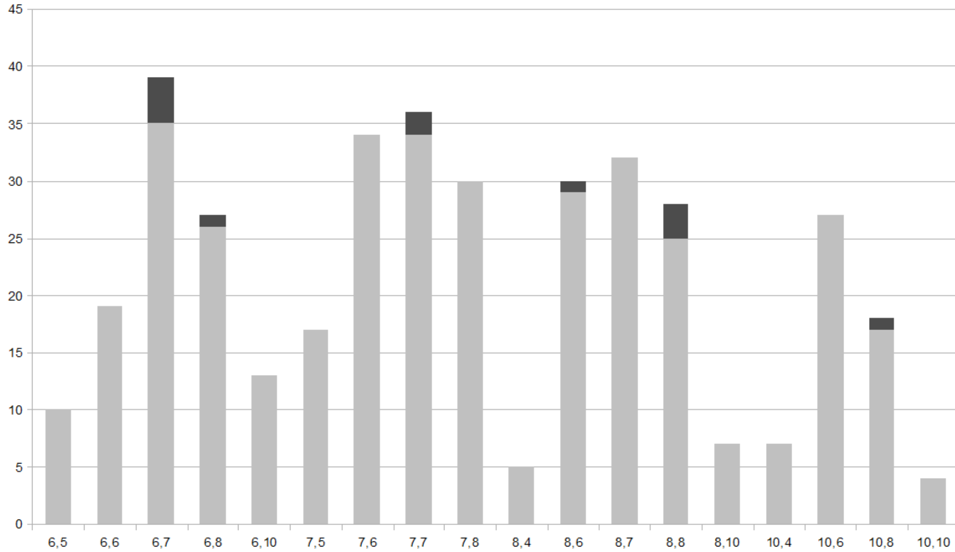

Figure 7: Experiments summary. x-axis: `B,T` parameters. y-axis: Light gray: the number of "good" computations (see Section 5.2) with the corresponding parameters. Dark gray heights: the number of fastest computations found with that parameter.

Results obtained by enabling the learning phase. A-B means main procedure and learning on A, unit-propagation on B. All computations on the hardware U. Running times in seconds.

Name	bwlarge.a	bwlarge.b	uf150-01	uf150-02	hanoi4	qg6-10	qg7-11	logistics.c
# Learned clauses	16	61	328	5083	813	247	5018	4528
CPU-CPU	0.46	7.66	0.77	59.64	15.63	24.85	1568	138
CPU-GPU	0.16	1.25	0.76	20.93	5.57	2.85	175	25.1
Speed up	2.9	6.1	1.0	2.8	2.8	8.7	8.9	5.5
GPU-GPU	0.12	1.02	0.60	18.67	4.88	2.64	167	21.66
Speed up	3.8	7.5	1.3	3.2	3.2	9.4	9.4	6.4

formulae back to the host is balanced by the more effective unit-propagation performed. As future work, we will explore how clause learning can be realized directly on the device, in order to move data within the GPU and to exploit a parallel learning schema.

These instances can be solved very quickly (in less than a second) by state-of-the-art SAT solvers. The aim of these tests is, thus, not to compete with these solvers, but to understand the potential benefits in instrumenting a parallel unit-propagation phase for execution on GPU platforms (which corresponds to selecting the mode 1 instead of mode 0, as discussed in Section 5.1). Moreover, our results suggest that the approach of delegating a parallel version of unit-propagation to the device can be realized by modifying the sophisticated unit-propagation procedure found in *any* of the existing SAT solvers. As future work, we plan to port to CUDA some simple yet powerful SAT solvers, such as MiniSat.

5.4 Enhancing Communication between CPU and GPU

A drawback of the conflict learning strategy is that, whenever a conflict is analyzed, a new clause is potentially added to the formula. If the conflict learning procedure is executed on the CPU, the device must be informed about new clauses, using a memory content exchange. This operation represents a bottleneck. We designed a version of our system to test whether a complete device implementation can offer some improvements. The only caveat is that the unit-propagation and learning phases require a synchronization that can be performed only by interleaving two host kernel calls. The device does not support native block-synchronization primitives. Therefore, it is not trivial to design a complete parallel scheme within the kernel.

Since our current parallel version runs both unit-propagation and learning on the GPU (using two distinct kernels), we opted to also move the tree expansion and backjumping control into the kernel that performs the

learning. This is realized by allocating the stack and tree data structures in the global memory of the device. In the current version of the solver, the code for learning executed by each thread is the same as the one executed on the host, i.e., it runs on a single block and single thread. This is a first attempt, and the investigation of the benefits of *truly parallel* learning schemes is one of our future works.

The host is in charge of the synchronization of the kernels, as they need to be properly interleaved. In this version, there is virtually no communication between the host and the device, except for a termination flag sent regularly from the device to the host. Despite the slower global memory access offered by the device during the tree expansion and backtracking, the time saved from copying the clauses from the host to the device provides a speedup of 25%. The results are reported in Figure 4, curve U.1(*), where the code is executed completely in the GPU, with learning deactivated, and in Table 5.3, last two rows, with learning activated.

5.5 Thread-Oriented Learning and Watched Literals

We implemented two additional variants of the system discussed in the previous sections, to allow us to expand the comparisons of mode 0 with mode 2.

In the first variant, we have added learning capabilities to mode 2. The idea is to allow each thread to perform its own local learning (within the GPU). This perspective is supported by our experiments, which show that the deterministic assignment of several variables at the beginning of the thread-level search phase frequently leads to learned clauses that contain at least one of such "deterministic" variables. This makes the learned clauses useless to the other threads. Thus, keeping learning local to the thread results in a smarter low-level search. On the other hand, the price of this is the increased frequency of access to the CUDA global memory (by the learning algorithm and to store/retrieve the new clauses). This cost may significantly reduce the benefits of having more cores. In the tests, we have also experimented with the use of the faster "shared" memory (used as single thread local memory) to store the data structures used by the learning phase. The code using shared memory proved to be slightly faster than the one using global memory (on average 10% faster).

In the second variant, we modified modes 0 and 2 by introducing *watched literals* [6]. Let us consider the case of a formula not containing any unit clauses. In each clause, we select exactly two literals (*watched*). As soon as one of them is falsified by a non-deterministic assignment or by unit propagation, another watched literal is identified within the same clause. If there are no literals left, then the other watched literal will trigger an immediate unit propagation. The watched literals do not need to be restored during backtracking. In some implementations (e.g., MiniSat [20]), the watched literals are the first two of each clause, and their updates are handled by reordering the literals in the clause. This approach does not fit well with a parallel scheme, where different threads may concurrently access the same formula. Our implementation exploits watched literals locally within each thread, by maintaining two pointers for each clause (along with some additional data structures to allow direct and inverse addressing).

The experiments show the same trends as those observed in the case of learning. The more threads we use, the faster the code performs. However, the maximum number of threads allowed is lower w.r.t. the numbers shown in the previous tests, due to the size of the memory local to each thread. This fact, and the large access to local data makes the speedup w.r.t. the sequential version negligible. The code and some additional observations can be found at `http://clp.dimi.uniud.it/sw/cudasat/`.

6 Related work

The literature on parallel SAT solving is extensive. Existing approaches tend to either tackle the approach from the point of view of parallelizing the search process—by dividing the search space of possible variable assignments among processors (e.g., [37, 7, 14, 46, 23, 31, 36])—or through the parallelization of other aspects of the resolution process, such as parallelization of the space of parameters, as in the portfolio methods (e.g., [25]).

Parallel exploration of the search space is, in principle, very appealing. The exploration of any subtree of the search space is theoretically independent from other disjoint subtrees; thus, one can envision their concurrent explorations without the need to communicate or extensive synchronizations. Techniques for dynamically distributing parts of the search tree among concurrent SAT solvers have been devised and implemented, e.g., based on *guiding paths* [46] to describe parts of the search space and different solutions to support dynamic load balancing (e.g., synchronous broadcasts [8], master-slave organizations [46]).

Sequential SAT solvers gain competitive performance through the use of techniques like clause learning [37, 47]—and this prompted researchers to explore techniques for sharing learned clauses among concurrent

computations; restricted forms of sharing of clauses have been introduced in [7, 15], while more general clause sharing have been used in [14, 34].

The closest proposals to the one we described in this paper are [38, 35, 24].

The authors of [38] focus on the fact that, on random instances, high parallelism works better than good heuristics (or learning). The authors focus on 3SAT instances—since any SAT instance can be rewritten into an equi-satisfiable 3SAT instance; they use a strategy that leads to a worst-case complexity of $O(1.84^n)$ instead of the standard $O(2^n)$ (let us notice, however, that n increases when converting a formula in the 3SAT format). Intuitively, given a clause $\ell_1 \vee \ell_2 \vee \ell_3$ a non-deterministic computation will explore three choices: one where $\ell_1 = \texttt{true}$, one where $\ell_1 = \texttt{false}$ and $\ell_2 = \texttt{true}$ and, finally, a third one where $\ell_1 = \ell_2 = \texttt{false}$ and $\ell_3 = \texttt{true}$. The idea is to use CUDA to recursively parallelize the three tasks, until the formula becomes (trivially) satisfiable or unsatisfiable. They use a leftmost strategy for selecting the next clause. The paper tests this approach on 100 satisfiable random instances, with 75 variables and 325 clauses. The paper lacks of a comparison w.r.t. a sequential implementation, therefore it is difficult to estimate the speedups obtained. We executed our code on the same instances used in [38] and observed execution times barely notable for all instances (being just too simple).

The proposal in [35] implements the so-called *survey propagation* using GPUs. Survey propagation is an incomplete technique which has proved to be effective on large 3SAT instances. Basically, a graph (called *factor graph*) with two kinds of nodes (for variables and clauses) and two kinds of edges (positive and negative) is built. A random initial labeling with floating point values from 0 to 1 is assigned to each edge. This labeling is continuously modified using some rules inspired from statistical physics and might converge to a solution. The updating of these values can be made by different independent threads and therefore naturally implemented on GPUs. In [24] the authors embedded the above idea, viewed as a variable order heuristics, into MiniSat. Being just a heuristics, MiniSat then tries all other possible assignments. The approach is proved to be effective for several input instances.

In the literature other proposals that relate to SAT solving and CUDA are found in [22, 12, 29, 4]. In [22] the authors focus on 3SAT problems and present a parallelization of the Boolean Constraint Propagation (Unit propagation procedure in our paper) through a partitioning schema which assigns a subset of the formula to each of the available MPs. In our work, we extend their idea of partitioning a set of clauses and distributing them among blocks of threads, in order to handle variable clause lengths. Moreover, we tested different configurations and optimized the number of blocks to be processed by GPU's cores. In [4] TWSAT heuristic is parallelized for selecting the best literal to be expanded in the search and the strategy runs multiple copies of the heuristic on the GPU. The results are collected by the CPU and given to MiniSat for the selection of the next literal expansion. In [29] the authors perform a brute force search on GPU by testing each of the 2^n combinatorial explosion of variables assignments, which has limited application for real test cases.

Other work has been done for incomplete SAT solvers, mainly based on local search and genetic algorithms. The paper [39] presents an approach for MAX-SAT solving in which GPU processing is applied to genetic algorithms and local search. In [12] the authors implements a P-system, capable of modeling NP-complete problems, and they specialize to the resolution of SAT problems. The approach is rather different from the DPLL resolution and it contains some brute force exploration of the solution space. The resolution explores subspaces of the population and it combines successful partial assignments until a conflict-less complete solution is found.

The work presented in this paper is an extension and generalization of the preliminary results presented in [17]. Other recent attempts to combine CUDA for solving combinatorial problems are found in [45] we show a preliminary implementation of an Answer Set Programming solver exploiting CUDA, in [9] we implemented a multi-agent solver for protein structure prediction exploting CUDA parallelism for local search, in [10] we presented a first general constraint solver running on CUDA and in [11] we built on in by showing the power of CUDA parallelism for implementing Large Neighborhood Search.

7 Conclusions and Future Work

In this paper, we discussed a preliminary exploration of the potential benefits of using a GPU approach to solve satisfiability (SAT) problems. SAT problems are notoriously challenging from the computational perspective, yet essential in a wide variety of application domains.

Mapping SAT to a GPU platform is not a trivial task—and understanding the benefits and pitfalls of such mapping is critical in moving parallel SAT from traditional architectures to GPUs. SAT problems require much less data and much more (non-deterministic) computation with respect to the traditional image and/or matrix

processing for which GPUs are commonly employed. With this seminal exploration, we demonstrated that, in spite of the different nature of the SAT problem, there are scenarios where GPUs can significantly enhance performance of SAT solvers. We explored and discussed alternative designs, providing two scenarios where the GPU can play a relevant role in the parallel computation. In the first one (namely, mode 2), parallel and divergent computations are launched. This attempt breaks the classical “rules” of GPU computing. The main reason for this counter-intuitive result is that the size of data involved and the random access pattern do not penalize the parallel execution. Moreover, this approach tries to exploit the presence of hundreds of processors that can perform independent tasks in parallel.

The second scenario (namely, mode 1) considers large formulae, typically generated after some form of clause learning, that can be efficiently processed by a GPU. This large amount of data can exploit all the benefits from non-divergent threads and structured memory accesses, thus providing significant and consistent speedups. We also believe that the creation of opportunities to handle a greater number of learned clauses is also beneficial to the exploration of the search space, which could potentially lead to a larger pruning of the search tree.

In our future work, we will explore a parallel implementation of clause learning techniques exploiting GPU and investigate the parallelization of a competitive and sequential SAT solver (such as MiniSat [20]). We will also explore scenarios where GPU-level parallelism interacts with CPU-level parallelism.

Acknowledgements

This research is partially supported by the projects INdAM-GNCS 2010, 2011, and 2014, PRIN 20089M932N, NSF 0947465, and by NMSU Manasse award. We would like to thank our students Luca Da Rin Fioretto, Francesco Peloi, Monica Pitt, and Flavio Vella, for their help in some parts of the implementation and testing. Finally, we would like to thank Federico Bergenti and Gianfranco Rossi for useful discussions.

References

[1] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2009.

[2] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2010.

[3] R.J.J. Bayardo and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *14th National Conference on Artificial Intelligence.* AAAI Press, 1997.

[4] Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, and Joost Vennekens. Parallel SAT-solving with OpenCL. In Hans Weghorn, Leonardo Azevedo, and Pedro Isaias, editors, *Proceedings of the IADIS International Conference on Applied Computing*, pages 435–441. IADIS, 2011.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation, Santa Clara, CA, 2008.

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

[7] W. Blochinger, C. Sinz, and W. Kuchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[8] M. Bohm and E. Speckenmeyer. A fast parallel SAT solver: Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[9] F. Campeotto, A. Dovier, and E. Pontelli. Protein structure prediction on GPU: a declarative approach in a multi-agent framework. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 474–479, Washington, DC, 2013. IEEE Press.

[10] Federico Campeotto, Alessandro Dal Palù, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of GPUs in constraint solving. In *Proceedings of Practical Aspects of Declarative Languages (PADL)*, volume 8324 of *Lecture Notes in Computer Science*, pages 152–167, New York, 2014. Springer.

[11] Federico Campeotto, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. A gpu implementation of large neighborhood search for solving constraint optimization problems. In *Proc. of ECAI 2014*.

[12] J.M. Cecilia, J.M. Garcia, G.D. Guerrero, M.A. Martinez del Amor, I. Perez-Hurtado, and M.J. Perez-Jimenez. Simulating a P system based efficient solution to SAT by using GPUs. *The Journal of Logic and Algebraic Programming*, 79(6):317 – 325, 2010.

[13] Imen Chakroun, Mohand-Said Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.

[14] W. Chrabakh and R. Wolski. GrADSAT: A parallel SAT solver for the GRID. Technical Report 2003-05, University of California, Santa Clara, CA, 2003.

[15] G. Chu, P. Stuckey, and A. Harwood. Pminisat: A parallelization of Minisat 2.0. Technical report, SAT-Race System Descriptions. Affiliated with SAT'08, Guangzhou, P. R. China, 2008.

[16] K. Claessen, N. Een, M. Sheeran, N. Sorensson, A. Voronov, and K. Akesson. SAT-solving in practice. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.

[17] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Exploiting unexploited computing resources for computational logics. In *Proceedings of the 9th Italian Convention on Computational Logic*, volume 857 of *CEUR Workshop Proceedings*, pages 74–88, Aachen, Germany, 2012. CEUR-WS.org.

[18] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[19] Gregory Frederick Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–488, New York, NY, 2011. ACM Press.

[20] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 502–518, New York, NY, 2004. Springer.

[21] Y. Feldman, N. Deshowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 75–90, Philadelphia, PA, 2004. Elsevier.

[22] Hironori Fujii and Noriyuki Fujimoto. GPU acceleration of BCP procedure for SAT algorithms. *IPSJ SIG Notes*, 2012(8):1–6, 2012.

[23] L. Gil, P. Flores, and L. Silveira. PMSat: A parallel version of Minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.

[24] K. Gulati and S. P. Khatri. Boolean satisfiability on a graphics processor. In *Great Lakes Symposium on VLSI*, pages 123–126, New York, NY, 2010. ACM Press.

[25] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[26] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 3:1–3:8, New York, NY, 2011. ACM Press.

[27] M. Henz and T. Muller. An overview of finite domain constraint programming. In *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies (APORS)*, Singapore, 2000.

[28] S. Holldobler, N. Manthey, V.H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel SAT-solvers. In *Advanced Computer Science and Information System (ICACSIS)*, pages 201–206, Washington, DC, 2011. IEEE Computer Society.

[29] Saiyedul Islam, Raghav Tandon, Shashank Singh, and Alok Misra. A highly scalable solution of an NP-complete problem using CUDA. In *Parallel Computing in Electrical Engineering (PARELEC), 6th International Symposium*, pages 93–98, Washington, DC, 2011. IEEE.

[30] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[31] B. Jurkowiak, C. Li, and G. Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

[32] B. Jurkowiak, C.M. Li, and G. Utard. Parallelizing Satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.

[33] David B. Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors. A Hands-on Approach.* Morgan Kaufmann/Elsevier, Philadelphia, PA, 2010.

[34] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *12th Asia and South Pacific Design Automation Conference*, pages 926–931, Washington, DC, 2007. IEEE Computer Society.

[35] Panagiotis Manolios and Yimin Zhang. Implementing survey propagation on graphics processing units. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 311–324, New York, NY, 2006. Springer.

[36] N. Manthey. Parallel SAT solving — using more cores. Technical Report KRR Report 11-02, Technische Universität, Dresden. Germany, 2011.

[37] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[38] Quirin Meyer, Fabian Schonfeld, Marc Stamminger, and Rolf Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. In Waleed W. Smari and John P. McIntire, editors, *2010 International Conference on High Performance Computing and Simulation (HPCS)*, pages 306–313, Washington, DC, 2010. IEEE Computer Society.

[39] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Hybrid of genetic algorithm and local search to solve MAX-SAT problem using NVIDIA CUDA framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, 2009.

[40] J. Nickolls and W.J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, 2010.

[41] NVIDIA. *NVIDIA CUDA C: Programming Guide.* NVIDIA Press, Santa Clara, CA, 2012.

[42] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[43] T. Schubert, M. Lewis, and B. Becker. PaMira: A parallel SAT solver with knowledge sharing. In *International Workshop on Microprocessor Test and Verification (MTV)*, pages 29–36, Washington, DC, 2005. IEEE.

[44] D. Singer. Parallel resolution of the satisfiability problem: A survey. Technical Report 2007-101, Université Paul Verlaine, Metz, France, 2007.

[45] Flavio Vella, Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@ASP: Experimenting with GPGPUs in ASP solving. In *Proceedings of the 28th Italian Conference on Computational Logic*, volume 1068 of *CEUR Workshop Proceedings*, pages 163–177, Aachen, Germany, 2013. CEUR-WS.org.

[46] H. Zhang, M.P. Bonacina, and J. Hsiang. PSato: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[47] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, Washington, DC, 2001. IEEE.