Exploiting Unexploited Computing Resources for Computational Logics*

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

[1] Università di Parma, alessandro.dalpalu@unipr.it
[2] Università di Udine, agostino.dovier@uniud.it
[3] Università di Perugia, formis@dmi.unipg.it
[4] New Mexico State University, epontell@cs.nmsu.edu

Abstract. We present an investigation of the use of GPGPU techniques to parallelize the execution of a *satisfiability solver*, based on the traditional DPLL procedure—which, in spite of its simplicity, still represents the core of the most competitive solvers. The investigation tackles some interesting problems, including the use of a predominantly data-parallel architecture, like NVIDIA's CUDA platform, for the execution of relatively "heavy" threads, associated to traditionally sequential computations (e.g., unit propagation), non-deterministic computations (e.g., variable splitting), and meta-heuristics to guide search. Experimentation confirms the potential for significant speedups from the use of GPGPUs, even with relatively simple modifications to the structure of the DPLL procedures—which should facilitate the porting of such ideas to other DPLL-based solvers.

1 Introduction

Propositional Satisfiability Testing (also known as *SAT solving* or, simply, *SAT*) [7] is the problem of determining whether a propositional logical formula T, typically expressed in *conjunctive normal form*, is satisfiable. That is, the problem is to determine whether there exists at least one assignment of Boolean values to the logical variables present in T such that the formula evaluates to *TRUE*.

SAT solving is a fundamental problem in computer science. From the theoretical perspective, the SAT decision problem was the first decision problem to be proved to be NP-Complete—thus, representing a cornerstone of computational complexity theory. SAT represents also a core practical problem underlying applications in a variety of domains, such as computational biology [8], verification and model checking [18], circuit design [19], and various forms of theorem proving and logic programming (e.g., [14, 11]). The widespread practical use of SAT has prompted extensive research in the development of efficient algorithms and data structures for SAT solvers—enabling the resolution of problems with thousands of variables and millions of formula components (e.g., as demonstrated by the results in recent SAT competitions [1]).

Even in the face of these impressive accomplishments, there are several classes of problems that are out of reach of the available solvers—in particular, there is evidence

* This work is partially supported by GNCS-11, PRIN 20089M932N, NSF 0812267, NSF 0947465. We would like to thank our students and collaborators Luca Da Rin Fioretto, Francesco Peloi, and Gabriele Savio for their help in the implementation and testing. We would like to thank also Federico Bergenti and Gianfranco Rossi for the useful discussions.

in the literature [12] showing that even the most sophisticated solvers have limits in scalability, and it is unclear whether such limits can be addressed by simple algorithmic and heuristic enhancements. This state of affairs has pushed research towards the exploration of alternative approaches—in particular, fueling research in the domain of *parallel* SAT solving. Parallel SAT solving takes advantage of architectures with multiple processors to speed up the exploration of the search space of alternative variable assignments. 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., [20, 3, 5, 21, 10, 13])—or through the parallelization of other aspects of the resolution process, such as parallelization of the space of parameters, as in portfolio methods (e.g., [12]).

The work discussed in this paper is aligned with the literature on parallel SAT solving—specifically with the approaches aimed at parallelizing the exploration of the search space by distributing it among concurrent computations. The novelty of our approach lies on the use of a *GPGPU (General Purpose computation on Graphics Processing Units)* approach to tackle the problem. This line of work is interesting under two orthogonal perspectives:

Opportunity: Traditional approaches to parallel SAT rely on general purpose architectures—i.e., either *tightly coupled* architectures (e.g., multicore or other forms of shared memory platforms) or *loosely coupled* architectures (e.g., Beowulf clusters). The former are widely available, the availability of shared memory enables a relatively simpler parallelization (e.g., it is easier to derive effective dynamic load balancing solutions), but it limits scalability. The latter offer greater opportunities to scale to several hundreds of cores, but are more expensive, relatively less commonly available, and the communication costs require significantly more expensive tradeoffs in terms of load balancing. GPGPU overcomes both problems—GPUs provide access to large numbers of processing elements and they are widely available in commodity architectures.

Challenge: 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. On the other hand, GPUs have been predominantly used for data parallel and SIMD-style computations. The mapping of a parallel exploration of the SAT search space on GPUs is the challenge we address.

We propose a first approach to this problem. We present the design of a GPU-based version of the *Davis-Putnam-Logemann-Loveland (DPLL)* procedure [9]—DPLL is at the core of the majority of existing SAT solvers. The proposed solution builds on the specific computational and memory model provided by NVIDIA's CUDA architecture.

2 CUDA

CUDA is the acronym of *Compute Unified Device Architecture*. This framework for parallel computing allows one to exploit GPU's capabilities for general purpose programs on NVIDIA architectures. For an introduction to CUDA programming we suggest [15].

We provide here a brief overview, adequate to allow the understanding of the following sections of this paper.

Host, global, device. A CUDA program is a high-level program that can be partially run on a GPU. The computer CPU (from now on, called *host*) and the (general purpose) Graphics Process Unit (or GPU, from now on, called *device*) can be controlled by a CUDA program that can take advantage of both resources. We refer to C CUDA programs only—other programming languages may involve a slightly different syntax. The code is handled by the `nvcc` compiler, which is in charge of mapping GPU parallel tasks to the video card.

Every function in a CUDA program is labeled as `host` (the default), `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. `global` functions are also called *CUDA kernel functions*. A `device` function can be called only by a function running on the device (`global` or `device`) and it runs on the device. 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.

Grids, blocks, threads. A general purpose GPU is a parallel machine. The hardware architecture contains a set of multiprocessors and each of them can run a number of parallel threads. The memory hierarchy is rather different from the traditional CPU multi-core structure (details in the following paragraph). 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 invoked 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 are they logically organized. The organization is hierarchical. 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. When mapping a kernel to a specific GPU, the number of processors and threads per processor may vary. In order to take advantage of multiple processors, the original kernel blocks are scheduled over multi processors and each group of threads in a block (typically 32) is run as a *warp* which is the minimal work unit on the device. Large tasks are encouraged to be fragmented over thousands of blocks and threads, while the GPU scheduler provides a lightweight switch between them. Using a `struct` data-type, the programmer can choose the number of blocks and the number of threads per block. Moreover, (s)he can also decide how to address the blocks (in one or two dimensions) and the threads in the block (in one, two, or three dimensions). For instance, if the programmer states that `dimGrid(8,4)` and `dimBlock(2,4,8)`, then the grid contains in total $8 \cdot 4 \cdot 2 \cdot 4 \cdot 8 = 2048$ threads. Blocks are indexed by $(0,0), (0,1), \ldots, (0,3), (1,0), \ldots, (7,3)$, while each block is associated to threads indexed by $(0,0,0), (0,0,1), \ldots, (1,3,7)$. 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 these 5 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, matrix operations, histograms 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. Divergent branches (e.g., the two if's bodies) are split into different warps, which may reduce the number of concurrent threads and affect the overall processor's *occupancy* (the ratio of active threads in a warp). The extreme case happens when each thread performs different operations.

Memories. Usually, host and device are distinct hardware units, with different purposes. Save for very cheap GPU used by some laptops, their memories are physically distinct. GPU's memories are connected to CPU's memory by a system bus, and DMA transfers can be used under some specific conditions (e.g., use of not paged allocation in RAM). When a host function calls a kernel function, the data required as input by the kernel computation need to be placed on the device. Depending on the type of GPU memory, different techniques are used, but, essentially, there is an implicitly (through memory mapping) and/or explicitly (through `cudaMemcpy`) 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 types of memories, 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. In particular registers and local memory have a thread life span and visibility, while shared memory has a block scope (to facilitate thread cooperation) and the others are permanent and visible from host and every thread on the device. 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 greater speedups, since access time and bandwidth of transfers are strongly affected by the type of memory and the sequence of access patterns by threads (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.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 (i.e., it has an access speed comparable to global memory). In the currently available GPUs, the constant memory is limited to 64KB and shared memory is limited to 48KB per block. In our first tests, we made use of constant memories for storing the input SAT formula; we quickly abandoned this approach, since even the simplest SAT input formulae easily exceeded 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. 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 and access pattern. For an interesting discussion of memory issues and data representations in a particular application (sparse matrix-vector

product) the reader is referred to [2]. Just a final pragmatic note: default installation parameters of GPUs include a timeout for a thread execution (typically, 2 seconds). For intensive applications, such as those presented in this paper, the option must be disabled (or, at least, its value must be significantly increased).

3 Davis, Putnam, Logemann, Loveland

We briefly outline the DPLL algorithm [9], here viewed as an algorithm to verify the satisfiability of a propositional logical formula in *conjunctive normal form (CNF)*. Let us recall some simple 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 $\ell_1 \vee \cdots \vee \ell_k$—often expressed simply as the set of its composing literals $\{\ell_1, \ldots, \ell_k\}$, 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`. If no such assignment exists, then the result of the search should be `false`. Fig. 1 provides the pseudo-code of the DPLL algorithm; the two parameters represent the CNF formula Φ under consideration and the variable assignment θ computed so far (at the first call, we set $\theta = [\,]$, denoting the empty assignment). The procedure `unit_propagation` 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. It returns the extended substitution θ' (possibly equal to θ). The procedure `ok` (resp. `ko`) returns `true` iff for each clause of the (partially instantiated) formula Φ there is a literal true (resp., there is a clause with all literals false). The procedure `select_variable` selects a not yet instantiated variable—using some heuristic strategy in making such choice, in order to guide the search towards a solution. The recursive calls implement the non-deterministic part of the procedure—known as *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 $\theta'[X/\texttt{true}]$. If this does not lead to a solution, then the assignment of `false` to X is attempted. Trying first `true` and then `false` as in Fig. 1 is another heuristics that can be changed.

The SAT community has developed a standard exchange data format for the encoding of problem instances, called the DIMACS format. Each clause is encoded on a separate line and terminated with 0. Each variable X_i is represented by the number i, its negative occurrences $\neg X_i$ by $-i$. The first line stores the number of variables and clauses. In Fig. 2, we provide a simple example.

4 A GPU implementation of DPLL

We present the main data structures and the overall structure of the CUDA program we developed. The program runs part of the DPLL computation on the host and part on the device. In Sect. 7, we provide details about some of the variants we attempted.

Formula and solution representation. A SAT instance is represented by:

```
DPLL(Φ,θ)
      θ′ ← unit_propagation(Φ,θ)
      if (ok(Φθ′)) return θ′
      else if (ko(Φθ′)) return false
      else X ← select_variable(Φ,θ)
            θ″ ← DPLL(Φ, θ′[X/true])
            if (θ″ ≠ false) return θ″
            else return DPLL(Φ, θ′[X/false])
```

Fig. 1. The pseudo-code for the DPLL procedure

```
p cnf 4 6
1 -2 3 0
1 4 -3 0
-1 2 3 4 0
2 3 -1 -4 0
-1 2 3 0
-1 2 -3 4 0
```

formula = |1|-2|3||1|4|-3||-1|2|3|4||2|3|-1|-4||-1|2|3||-1|2|-3|4|

clause_pointer = |0|3|6|10|14|17|(21)|

NV = 4 NL = 21 NC = 6

Fig. 2. A simple SAT instance in the DIMACS format and its low level representation

- a vector `formula` storing, sequentially, the literals in the clause,
- a vector `clause_pointer`, that stores the pointers to the beginning of the various clauses (it also stores a reference to the end of `formula`), and
- three variables `NV`, `NL`, and `NC`, that store the number of variables, of literals, and of clauses in the formula, respectively.

Fig. 2 shows and example of such representation. The variable assignment is stored in an array of integers, called `vars`, of size `1+NV`. The value of variable $i \in \{1, \ldots, \texttt{NV}\}$ is stored in `vars`$[i]$; 1 stands for `true`, 0 for `false`, and -1 for (still) unassigned.[5] The component `vars`$[0]$ is used as a Boolean flag.

Host code. The `main` host procedure reads a mode flag and the name of the DIMACS file containing the formula to be loaded. A first computation of the DPLL algorithm, implemented recursively in C, runs in the host. If the code is called with mode flag 0, the computation *proceeds in the host* until its termination. Otherwise, if the number of uninstantiated variables after unit propagation does not exceed a threshold (called `MaxV`, its typical range is 50–80), a kernel function is called.[6] For reducing the amount of data to be dealt with by the device code, the formula is filtered using the current partial assignment θ. Namely, all clauses satisfied by θ are deleted and all literals set to `false` by θ are removed from the remaining clauses. The filtered formula and an empty assignment for the remaining variables are passed to the device global memory.

The procedures `unit_propagation`, `ok`, and `ko` are defined as particular cases of the integer function `mask_prop`. Given a (partial) assignment θ stored in the `vars` array, `mask_prop` fills an array `mask` in such as way that for each clause i:

- `mask`$[i] = 0$ if clause i is satisfied by θ;

[5] Different negative values will be used for unassigned variables in block shared versions of `vars` for technical reasons, as explained in Sect. 4—Fig. 4.

[6] Unit propagation itself can be parallelized using CUDA. We discuss this in Sect. 7.

- `mask`$[i] = -1$ if all literals of the clause i are falsified by θ;
- `mask`$[i] = u$ if clause i is not yet satisfied by θ, and there are still $u > 0$ unassigned literals in it.

Furthermore, `mask_prop` returns -1 if there is a value of i such that `mask`$[i] = -1$; or 0 if for all i `mask`$[i] = 0$; or the smallest index of the clause with a minimum number of uninstantiated literals, otherwise. This information is used for implementing the `select_variable` function (see Fig. 1). The use of this approach allows us to introduce a selection strategy that is similar to the well-known *first-fail strategy*. Clauses that allow unit propagation are selected first. `mask_prop` is defined in such a way that it can be used either by the host or by the device (cf. Sect. 4) and it runs in linear time on the size of the formula. A faster incremental version has been also implemented; however, additional data structures on CUDA are required to be communicated. In this paper, we have preferred the simpler version implemented in the same way on the CPU and on the GPU, that prompts for a parallelization on the GPU (cf. Sect. 7).

Device code. In our implementation, we call the device by specifying `B` $\times$ `B` blocks, each addressing `T` $\times$ `T` threads. Typical values are `B` $= 64$ and `T` $= 16$. Let us focus, in what follows, on these two values, for the sake of simplicity.

```
...
dim3 blocks(B,B), threads(T,T);
...
CUDADPPL<<<blocks,threads>>>( PARMS )
...
```

In the parameters `PARMS` we pass all data relative to the filtered formula. Since a kernel function cannot return a result, we copy back the first cell of the device-version of the array of variables, and use it as a Boolean flag to signal satisfiability of the SAT instance. A shared vector `block_vars` is initialized using the (binary expansion of the) block coordinates, as done by the code in Fig. 3. The first $2\log($`B`$)$ variables are assigned deterministically in this way. Note that the code in Fig. 3 shows a simplification of the actual code, where a unit-propagation phase follows each assignment in order to deterministically propagate its effects. Then the task is delegated to each thread. We have tried to exploit at best the access times of the different device memories: The device computation uses the partial assignment stored in the (fast) array `block_vars` and the local (slow) array `delta_vars`.

Let us elaborate on this idea. Let us assume that there are 30 unassigned variables, and let us consider Fig. 4. The first 16 variables are assigned according to the block coordinates (we denote this with $\frac{0}{1}$ in the figure). The remaining variables are stored in a private thread vector `delta_vars` and the corresponding components of `block_vars` store pointers to them, as shown in Fig. 4. In particular, if `block_vars`$[i] = -j < 0$ then the value of variable j, currently handled by the thread, is stored in `delta_vars`$[j]$; these variables are initially set to -1, to encode the fact that no commitment has been made regarding their truth value yet. Notice that this representation of partial assignments is exploited by the procedure `mask_prop` mentioned earlier. In fact, in evaluating a given partial assignment on the input formula, whenever `mask_prop` processes a variable i occurring in a literal, it first checks the value of `block_vars`$[i]$. If this value

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

Fig. 3. Deterministic assignments of variables in the shared memory

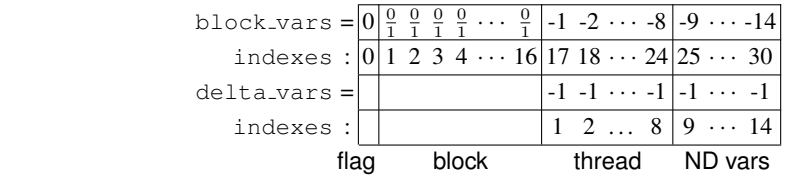

Fig. 4. Multilevel memory organization for variable assignments.

is positive or zero, then it assigns such a value to the variable (and hence, the proper value to the literal). Otherwise, it considers the value of $\texttt{delta\_vars}[-\texttt{block\_vars}[i]]$.

After these initial assignments of values have been completed, an iterative implementation of DPLL is called. It starts by calling a device version of `mask_prop`—the main difference being that this version does not explicitly allocates an array `mask` as the host version does. If the formula turns out to be satisfied, the whole assignment is returned. If a clause is made false by the current partial assignment, then either a backtracking phase is activated, locally to the single thread, or the thread terminates with failure, if no other choices are possible. When the formula is not satisfied and no falsified clause is found, the assignment is extended; if possible, deterministic unit propagation is applied or, otherwise, the splitting rule is activated. In each of the first 8 applications of the splitting rule, the choices of the values to be assigned to the selected variables are done according to thread's address. In particular, this is done as shown in Fig. 3, by using the binary expansion of the value of `threadIdx` in place of `blockIdx`. For the subsequent applications of the splitting rule, the first value attempted is the one that satisfies the first not yet satisfied clause. The other value will be attempted via backtracking.

Let us close 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 the maximum number of variables sent uninstantiated to the device, minus $2\log(\texttt{B})$—the number of variables assigned by exploiting the block coordinates. The stack elements contain 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.

5 Experiments

We tested a large variety of SAT instances taken from classical examples and recent competitions. We report details about two specific instances: the Pigeon-hole problem by John Hooker (72 vars, 297 clauses, 648 literals), called `hole8` and a permuted SAT Competition 2002 formula with seed=1856596811, called `x1_24` (70 vars, 186 clauses, 556 literals). In these experiments we are not concerned with the search strategy and heuristics. For this purpose, we implement the same simple variable selection described in Sect. 4 on both CPU and GPU versions. Clearly, if compared to cutting-edge solvers, the results reported here are some orders of magnitude slower. Our focus is on ability to extract parallelism from DPLL (and we need to remember that many of the existing solvers build on it).

Our aim is to compare only the benefits of a GPU search on a same tree structure. Therefore, heuristics about tree explorations should have a similar impact on CPU and GPU. Moreover, we expect that the parallelization of the heuristics on GPUs could be another key issue in our future work.

We also test the results on various devices, ranging from small scale GPUs up to top ones. Due to lack of space, we only report results from using a large scale platform. A remarkable note is that even with less capable devices, speedups are attainable, and thus these ideas are of an immediate application on standard workstations. Since the differences between CPU and GPU times are also dependent on the host CPU, the actual meaning of a relative speedup is a bit fuzzy. We however note that the CPU cores we used are of standard level for today's market (e.g., Xeon e5xxx, Intel i7), and thus this can provide a baseline for reference. The hardware used for the tests is the following:

- *Host:* a 12 core Xeon e5645, 2.4GHZ, 32GB RAM (used as a single core).
- *Device:* an NVIDIA Tesla C2075, 448 cores, 1.15GHz, bandwidth 150GB/s.

All the experiments have been conducted averaging the results over repeated runs. In Fig. 5, we report two rows of charts. Row A shows the results for the `hole8` problem. Each column shows the performance for different `MaxV` variables: 72 (number of variables for the problem), 70, and 65. Row B shows the results for the `x1_24` problem. The columns are associated to the following `MaxV`: 70 (number of variables for the problem), 68, and 65. In the x-axis are reported the number V of variables non-deterministically handled by each thread. This relates to the `T` parameter defined above as follows: $\texttt{T}^2 = 2^V$, since `T` is the square root of the number of threads that assign values to the set of variables V. Different lines in the graphs capture different number V' of variables handled by blocks. Again, this relates to the `B` parameter: $\texttt{B}^2 = 2^{V'}$.

Reducing `MaxV` with respect to the maximum number of variables causes the CPU to perform a significant amount of work in the top of the search tree. In practice, this does not introduce much benefit, since the CPU is outperformed by the GPU. However,

parallel unit propagation, with specific conditions (see Sect. 7) can introduce relevant speedup as well. In such cases, we expect that a balance between the two GPU usages could find an optimal tradeoff.

The number of threads per block 2^{10} ($V = 10$) is the maximum supported by the device; it is interesting to observe that this number does not always lead to the best performance results. This can be justified by the fact that an increase in the number of threads (and having many blocks) leads to an increase in the number of uncoalesced memory accesses and bank conflicts on the shared memory. Given the totally divergent thread codes, reducing the degree of concurrency represents the only tweaking. Clearly there is room for optimizations, in particular focusing on the memory access pattern, in order to bring the execution closer to an SIMD model, more suitable to the GPU. The number of blocks in use can affect the performance. In this example, a large number of blocks offers a large pool of tasks to the scheduler. Since many of the running blocks are suspended for memory accesses, it is important to have a large queue in order to keep the processors busy.

This first set of benchmark let us conclude that our implementation is capable of notable speedup w.r.t. CPU version. For `hole8` we record an execution time of 14.18s ($V = 6$, $V' = 16$, $\texttt{MaxV} = 72$), that, compared to 71.2s of CPU time, corresponds to a a speedup greater than 5. For `x1_24` we record a 26.42s ($V = 6$, $V' = 10$, $\texttt{MaxV} = 70$), that, compared to 1053s of CPU time, almost reaches a $40\times$ speedup.

Tests performed on different GPU platforms produce comparable results; the main difference is that the best performance results require different choices of parameters. In particular, we would like to spend a few words on one of the machine used that represents a typical office desktop configuration:

- *Host:* Intel core i7, Windows 7, 64 bits, 3.4–3.7GHz
- *Device:* NVIDIA GeForce GTX 560, 336 cores, 1.62-1.9GHz, bandwidth 128GB/s.

In this case the GPU is not dedicated to GPGPU computing, but it is also used to control the monitor. For `hole8`, we measured an average execution time of 4.43s ($V = 6$, $V' = 10$, $\texttt{MaxV} = 71$), that, compared to 34s of CPU time, generates a $7\times$ speedup. For `x1_24`, we observe an execution time of 21.76s ($V = 8$, $V' = 10$, $\texttt{MaxV} = 70$), that, compared to 324s of CPU time, gives a $15\times$ speedup.

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., [20, 3, 5, 21, 10, 13])—or through the parallelization of other aspects of the resolution process, such as parallelization of the space of parameters, as in portfolio methods [12].

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 for communication 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* [21] to describe parts of the search space and dif-

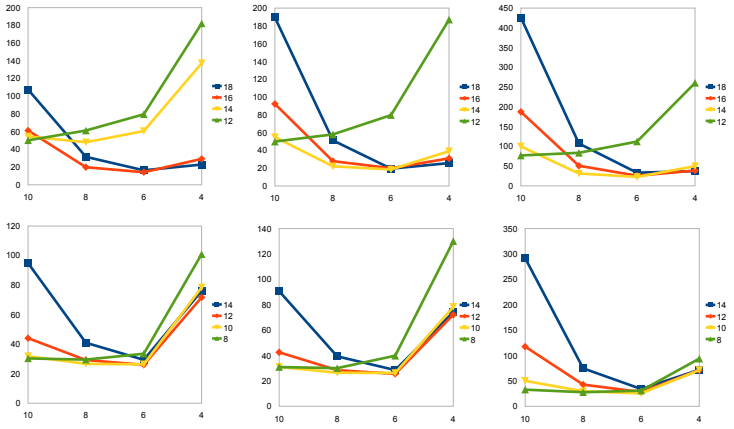

Fig. 5. GPU times with different parameters. Upper row, instance `hole8`: CPU time is 71.2 seconds. From left to right, `MaxV` is 72 (1 GPU call), 70 (3 GPU calls), 65 (8 GPU calls). Lower row, instance `x1_24`: CPU time is 1053 seconds. From left to right, `MaxV` is 70 (1 GPU call), 68 (2 GPU calls), 65 (8 GPU calls). All the times are in seconds.

ferent solutions to support dynamic load balancing (e.g., synchronous broadcasts [4], master-slave organizations [21]).

Sequential SAT solvers gain competitive performance through the use of techniques like clause learning [20, 22]—and this prompted researchers to explore techniques for sharing learned clauses among concurrent computations; restricted forms of sharing of clauses have been introduced in [3, 6], while more general clause sharing have been used in [5, 16]. The closest proposal to the one we describe in this paper is the one presented in [17]—being the only proposal dealing with SAT on GPU platforms. The authors focus on the fact that, on random instances, high parallelism works better than good heuristics (or learning). They consider 3SAT instances and 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). Basically, given a clause $\ell_1 \vee \ell_2 \vee \ell_3$ a non-deterministic computation starts with $\ell_1 = \texttt{true}$, another with $\ell_1 = \texttt{false}$ and $\ell_2 = \texttt{true}$ and, finally, a third with $\ell_1 = \ell_2 = \texttt{false}$ and $\ell_3 = \texttt{true}$. The idea is to use CUDA for 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 speedup they obtain. We run our parallel code on the same instances and observed execution times barely notable for all instances.

7 Discussion

In this section we report on some variations of our approach—most of which did not show any significant performance benefits. One can observe that mapping a search problem, described by a search tree, on a GPU platform is non-trivial. We identified two

main aspects that deserve to be parallelized: the operations performed on a single node (e.g., unit propagation) and the parallel exploration of non-deterministic branches of a specific depth (i.e., a set of independent choices is explored by a specific thread). The approach described in the paper is predominantly based on subtree parallelism at the very bottom of the search tree, while the upper part of the tree is handled by the CPU.

Parallelism within a node. We implemented a parallel version of the unit propagation, where the computation of unit variables is performed in parallel. We tested different worker/task mappings: a thread is in charge of checking a specific clause or it is in charge of checking a specific literal.

In the first case, the fact that each clause may have a different length causes some asymmetries in the computation, thus divergent branches may become an issue. We investigated some sparse matrix algorithms applied to matrix multiplication (cf., [2]). If the formula is represented as a sparse matrix, the dot product between a row (clause) and the variables represents the processing of the relevant assignments to variables in the clause. However, while the dot product simply adds up the contributions provided by corresponding pairs—our algorithm needs to handle conditional increments, associated to the value of the variables and the sign of the literals. We tested the *Compressed Sparse Row* implementation, which is very similar to the one in use by our program and we did observe a general poor performance. The main reason is that matrix multiplication is performed efficiently when a certain number of not null element are involved. Typically the structure of a clause involves a small subset of variables, which are less than the number of clauses. Therefore the mapping of a thread to a clause may end up with an unfavorable setting. Another reason for the poor performance is that the memory access pattern for reading the formula is rather unstructured and it might be the case that the same data have to be read by different threads.

Mapping threads to single literals of formula also performs poorly, since the amount of work for each thread does not compensate the overhead.

In what follows we move towards some ideas to increase the kernel's work load.

Parallelism on a subtree. In order to generate larger tasks for each thread, we decided to expand the size of the problem, by considering multiple assignments to the variables at the same time. This allows us to generate 2^k different problems, where k is the number of variables being checked. On the search tree this equates to the exploration of a complete subtree of depth k for a fixed set of variables. The bottleneck of this approach is that the amount of data to be passed back to the host is significant: the 2^k results are copied back to host before the next subtree can be explored. Moreover, the stack must hold at most 2^k alternative valid sub-branches. Moreover, this technique generates many failure branches because unit propagations frequently invalidates the made choices. Consequently, a large set of threads computes useless branches.

Complete GPU parallelism. This can be achieved with different design patterns. The base line is that the temporary data reside on the GPU's memory and there is no exchange with the CPU unless to communicate the formula and to retrieve the result. One possibility is to extend the application of the device code presented above, so that it can handle all the variables involved in the formula. This causes a single GPU call that handles the whole search tree. We noticed that this choice is optimal for current implementation. This trivial extension, however, suffers from the fact that the work of

each thread is completely independent (divergent) and many of the accesses to global and shared memory are not structured. Potentially, there is a good margin for optimization, prior a re-organization of the kernel. The idea is to keep a partition of the tree into subtrees described above. Every single subtree is computed by a kernel and it has to be interleaved by a kernel that takes care of the (iterative) tree expansion into subtrees and of the stack/backtracking issues. We tested a model where an iterative version of Depth First Search is implemented as a kernel based on GPU global memory. Another kernel takes care of propagation and variable selection strategy. An outer loop on the host controls the correct interleaving of the kernels. Here the not symmetric step is represented by the stack size that has to store all choice points discovered during the search and the unit propagated variable that may differ in number across the various branches.

Stacks. The stack that handles the recursion on the device is currently implemented by arrays that are mapped to global memory by the compiler. We tested a version in which this memory is mapped to shared memory, so that each thread accesses an independent portion. This version has two drawbacks: the first is that the large stack size and the limited shared memory size restrict the number of threads per block; the second is that performance is also affected, since the large use of shared memory restricts the occupancy of the processors.

Large instances. We observed how the actual work for single threads is not sufficient to generate interesting speedups. The idea to increase the work load by exploring different non-deterministic choices in parallel is promising, but also it appears difficult to be organized in a complete SIMD approach. Another potential target could be the parallel handling of extremely large formulae with few variables. While formulae of this kind are uncommon in typical SAT instances, they could be easily generated when clause learning techniques are used—as these may lead to redundant clauses that can easily grow up to millions. In this context, it is interesting to see the capability of a kernel for the computation of unit propagation. The most likely scenario is to have few literals per clause versus a large amount of clauses. The formula is partitioned according to the number of threads of each block and a certain number of blocks (experimentally the double of the processors available) is created. Each block is in charge of analyzing a subset of clauses. A block is dedicated to analyze a number of clauses, where each thread is mapped to each literal appearing in one of these clauses. Then, each block computes the best candidate through a logarithmic reduction schema and then iterates the process on the next set of clauses.

We implemented two equivalent versions for CPU and GPU and tested them by modifying the size of the input. Since we are interested in the performance of the single kernel and there is no dependence on choice points, we simply appended a formula an exponential number of times, in order to create some large formula to analyze. We tested various sizes: the smallest, named 0, with 186 clauses is the `x1_24` instance. The i-th instance is the concatenation of 2^i copies of the `x1_24` instance. The largest, named 16, contains 24.3M clauses. We selected 768 threads per block, which was the optimal value for the system in use (14 cores and capabilities 2.1). In Fig. 6 we depict the clause size versus the computational times for CPU and GPU (both in logarithmic scale). It can be seen that almost $10\times$ of speedup is possible. It is interesting to note that the cases 0–3 contain the typical sizes for hard problems (up to thousands clauses).

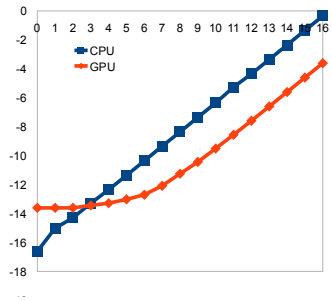

Fig. 6. CPU and GPU timings for unit propagation with large instances

However, in this range the GPU performs poorly, mainly because of the overhead of the GPU invocation and the relatively small task to be performed. This also explains the poor results achieved when we tried to the parallel version of the unit propagation in the top part of the tree. Last instance requires 300MB on the video card, and thus it may be considered one of the largest instances that could be handled by a low cost GPU.

8 Conclusions and Future Work

In this paper, we discussed a preliminary exploration of the potential benefits of using a GPGPU approach to solve satisfiability (SAT) problems.

Mapping SAT to a GPU platform is not a trivial task. 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. We demonstrated that, in spite of the different nature of the SAT problem, there are scenarios where GPUs can significantly enhance the performance of SAT solvers. We explored and discussed alternative designs, providing two scenarios where the GPU can play a relevant role. In the first one, 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 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 notable 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 the integration of clause learning techniques in the GPU code and investigate *(i)* how it affects the performance of the parallel solver, and *(ii)* how it can itself benefit from parallelization. We will also explore scenarios where GPU-level parallelism interacts with CPU-level parallelism.

References

[1] The International SAT Competitions. `http://www.satcompetition.org`, 2012.

[2] N. Bell and M. Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA, 2008.

[3] W. Blochinger, C. Sinz, and W. Kuchlin. Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning. *Parallel Computing*, 29(7):969–994, 2003.

[4] M. Bohm and E. Speckenmeyer. A Fast Parallel SAT Solver: Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996.

[5] W. Chrabakh and R. Wolski. GrADSAT: A Parallel SAT Solver for the GRID. Technical Report 2003-05, University of California Santa Barbara, 2003.

[6] G. Chu and P. Stuckey. Pminisat: A Parallelization of Minisat 2.0. Technical report, SAT-Race System Descriptions, 2008.

[7] K. Claessen, N. Een, M. Sheeran, N. Sorensson, A. Voronov, and K. Akesson. SAT-Solving in Practice. *Discrete Event Dynamic Systems*, 19(4), 2009.

[8] F. Corblin, L. Bordeaux, Y. Hamadi, E. Fanchon, and L. Trilling. A SAT-based Approach to Decipher Gene Regulatory Networks. In *Integrative Post-Genomics*, 2007.

[9] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] L. Gil, P. Flores, and L. Silveira. PMSat: A Parallel Version of Minisat. *J. on Satisfiability, Boolean Modeling and Computationa*, 6:71–98, 2008.

[11] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In *AAAI*, pages 61–66, 2004.

[12] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[13] B. Jurkowiak, C. Li, and G. Utard. A Parallelization Scheme Based on Work Stealing for a Class of SAT Solvers. volume 34, pages 73–101, 2005.

[14] H. Kautz and B. Selman. Planning as Satisfiability. In *European Conference on Artificial Intelligence*, pages 359–379, 1992.

[15] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann/Elsevier, 2010.

[16] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *12th Asia and South Pacific Design Automation Conference*. IEEE Computer Society, 2007.

[17] Q. Meyer, F. Schonfeld, M. Stamminger, and R. Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. In W. W. Smari and J. P. McIntire, editors, *HPCS*, pages 306–313. IEEE, 2010.

[18] M. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-based Formal Verification. *STTT*, 7:156–173, 2005.

[19] J. Silva. *Search Algorithms for Satisfiability Problems in Combinational Switching Circuits*. PhD thesis, The University of Michigan, 1995.

[20] J. P. M. Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[21] H. Zhang, M. Bonacina, and J. Hsiang. PSato: A Distributed Propositional Prover and its Application to Quasigroup Problems. *J. of Symb. Comput.*, 21, 1996.

[22] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *ICCAD*, pages 279–285, 2001.