3-SAT on CUDA: Towards a Massively Parallel SAT Solver

Quirin Meyer, Fabian Schönfeld, Marc Stamminger, Rolf Wanka
Department of Computer Science, University of Erlangen-Nuremberg, Germany
{quirin.meyer | marc.stamminger | rwanka}@cs.fau.de
fabian.schoenfeld@gmx.net

ABSTRACT

This work presents the design and implementation of a massively parallel 3-SAT solver, specifically targeting random problem instances. Our approach is deterministic and features very little communication overhead and basically no load-balancing cost at all. In the context of most current parallel SAT solvers running only on a handful of cores, we implemented our solver on Nvidia's CUDA platform, utilizing more than 200 parallel streaming processors, and employing several millions of threads to work through single problem instances. As most common sequential solver techniques had to be discarded, our approach is additionally supported by a new set of global heuristics, designed specifically to be easily exploited by the underlying thread parallelism.

KEYWORDS: GPGPU, thread level parallelism, load balancing and sharing, random 3-SAT.

1. INTRODUCTION

As parallel resources are becoming increasingly cheap to acquire, more and more established algorithms are being revised to take advantage of this, in order to boost their performance and/or gain new insights into the inner workings of long known problems. Together with the field of high performance computing, the realm of compute intensive algorithms is trying to make the most of this. Here many NP-complete problems are being investigated [9], with one of the most discussed being Boolean Satisfiability (SAT). It was the first problem to be proven NP-complete [4] and subject to different solving approaches for almost fifty years by now. It is studied intensively by theoreticians due to its proximity to a whole different class of problems (as 2-SAT is merely P-complete) and arguably being the most bare-bones NP-complete problem. In practice the problem retains its relevance by still occurring in many applications of current interest like theorem proving, FPGA routing, and Electronic Design Automation.

1.1. Previous Work: The Parallel Track

Most current high end SAT solvers are still sequential algorithms [7], [15], [16] based on extending the classic Davis-Putnam-Logemann-Loveland (DPLL) procedure [5], [6], by incorporating additional state data collected over the course of a sequential solving run. In recent years several successful parallel solvers have emerged, establishing a new "parallel track" in annual SAT competitions in 2008. This newer generation of solvers, however, is still mainly based on the core DPLL framework and the techniques developed for their sequential predecessors. PSATO [21] and Gradsat [2], [3] are two examples for high end parallel solvers, which consist of bringing their sequential counterparts ([20] and [15]) to a parallel platform. As with Nagsat [8], they are based on a master-slave approach, where the master runs a modified DPLL procedure and distributes sub-problems to the available parallel nodes - which again run their version of the DPLL algorithm. The current state of the art is represented by the *portfolio-based* ManySAT [10]: Its parallel approach consists of running several different (and carefully calibrated) sequential solvers in parallel, returning the first result obtained by any one of them.

As can be seen, the current parallel track is still mainly based on the advancements made during developing sequential solvers. The execution itself is either done on just a few local cores (34 for Gradsat, 29 for PSATO) or small scale grids with high communication costs and a comparatively small set of largely independent processing units. All their design decisions are of course validated by their success. The question for massive parallelism in SAT solving, however, still remains largely unanswered.

1.2. Enter GPGPU

While current parallel solvers are primarily implemented to run on computer clusters, an alternative test bed is

provided by GPGPU: *General Purpose computing on Graphics Processing Units*. As a lot of graphical tasks are inherently parallel, graphics hardware is designed to efficiently handle computations concurrently. GPGPU is the idea of utilizing this processing power to solve problems unrelated to computer graphics. In late 2006, Nvidia introduced its CUDA API [14], which allows for direct access to a CUDA enabled graphics card's parallel hardware, and thereby supplies developers with a highly efficient co-processor to handle specific workloads.

2. A PARALLEL CUDA SAT SOLVER

Since its introduction, CUDA is continuously developed further, with its latest incarnation being manifested in the Fermi chipset announced for 2010 [19]. CUDA offers massive thread parallelism without the need for an expensive computer cluster setup, and allows for the concurrent execution of hundreds of thousands of lightweight threads, running on several hundreds of CUDA *streaming processors* (around 200 on current graphics cards and 512 on Fermi). A batch of 32 threads is bundled within a so called *warp*, which denotes the amount of threads actually being physically executed in parallel on a single processor. On a larger scale, threads are part of a CUDA *thread block*, which denotes a bundle of threads operating on a set amount of shared memory. The parallel paradigm for CUDA is a block-wide SIMD model. That is, all threads of the same block are required to execute the same instruction, while different blocks may execute different functions. The execution of a function on the hardware itself is organized in so called CUDA *kernels*. A kernel simply denotes a procedure running on the graphics card - the *device*. Each kernel is invoked by the *host*, i.e., the computer containing the device, and operates on a grid of CUDA thread blocks, which is also defined by the host.

Besides its parallel architecture, the most defining CUDA aspects for our work are the constraints of the platform, primarily the limiting amount of on-chip memory. In essence, two distinct disadvantages present themselves: First, with the vast number of threads running in parallel, the remaining available space for additional state data per single thread becomes very limited. Essentially all modern SAT solver frameworks are used to employing quite a large amount of such state data, while dealing with a *single* "current" formula (like implication graphs, databases of learned clauses, and so on). Our massively parallel solver, however, deals with thousands of search paths at once, and the hardware simply does not provide the memory resources which would allow the same amount of state data per search path.

The second memory induced issue is much more straightforward: Less space means shorter formulas. Note that current generation SAT solvers often do not even use the term "formula" when addressing a problem instance, but rather refer to it as "clause database", due to the impressive length of formulas current solvers are able to deal with. Not so with limited CUDA resources, however, which results in compatible CUDA solver formulas being of a much more compact format.

Luckily, both of these memory issues push our solver in the same direction: random 3-SAT instances. Current solver advances allow for industrial problems containing more than hundreds of thousands of clauses and variables to be solved in mere minutes, due to the inherent and exploitable structures encoded within such instances. Purely random SAT instances, however, lack such internal structure, and consequently state hard problems via much smaller formulas. Furthermore, it has been discovered that modern indispensible solver techniques such as clause learning and complex state dependent heuristics, like conflict graphs, often degenerate to brute-force style rules when handling chaotic random instances - for example by regularly backtracking to the last decision made, instead of some more meaningful decisions higher up in the search tree. Consequently, parallel SAT solving seems to be well suited to handling random instances, as their smaller size reduces the amount of memory required to handle them, and common memory intensive techniques are additionally not guaranteed to deliver the same performance boosts as they do in the case of industrial (i.e., structured) problems - and may thus be discarded without impacting the overall solver too much.

2.1. The Base Algorithm

The D&C-3SAT algorithm parallelized and brought to CUDA in this work employs a divide-and-conquer approach for solving 3-SAT instances, as is depicted by Algorithm 1. It takes a formula f in *Conjunctive Normal Form* (CNF) over n Boolean variables and r clauses, with each clause consisting of at most three literals (the CNF format denotes a conjunction of clauses, with each clause being a disjunction of literals). The algorithm is described in detail in [13] and is a simplified version of the algorithm presented in [17]. Besides providing a framework suitable to be easily parallelized, the algorithm also reduces the worst-case time complexity from $O(r \cdot 2^n)$ (the brute force approach) to $O(r \cdot 1.84^n)$ [14, pp. 171ff].

The core idea of this approach is to exploit the fact that all possibilities to satisfy any given clause can be broken down to three different cases: Initially, the first literal is set to true, which either contributes to solving the formula or the subsequent recursion returns a contradiction. In the latter case, setting the first literal to true obviously did not work, and thus the corresponding variable is set to falsify

the literal. In order to satisfy the clause now, the second literal is set to true, and the process is repeated. In case this again leads to a conflict, the final, third approach is taken by setting the first two literals to false and only the last one to true.

Algorithm 1: Divide & Conquer SAT Solver

```
   Data: Formula f in CNF
1  if length(f) < limit then return brute_force_solve(f);
2  Clause c ← pick_random_clause(f);
3  // Note: c consists at most of literals lit_a, lit_b, lit_c;
4  if literals_contained_in(c) equals 1 then
5  |  Formula f_1←apply(f, lit_a=true);
6  |  return D&C-3SAT(f_1);
7  end
8  if literals_contained_in(c) equals 2 then
10 |  Formula f_1←apply(f, lit_a=true);
11 |  Formula f_2←apply(f, lit_a=false, lit_b=true);
12 |  return D&C-3SAT(f_1) OR D&C-3SAT(f_2);
13 end
14 if literals_contained_in(c) equals 3 then
15 |  Formula f_1←apply(f, lit_a=true);
16 |  Formula f_2←apply(f, lit_a=false, lit_b=true);
17 |  Formula f_3←apply(f, lit_a=false, lit_b=false, lit_c=true);
18 |  return D&C-3SAT(f_1) OR D&C-3SAT(f_2)
   |           OR D&C-3SAT(f_3);
19 end
```

Thus, in every step of the recursion some clause is selected, and - depending on the number of literals it carries - the search expands in up to three new directions, following all possibilities to satisfy the chosen clause. The recursion is terminated as soon as a sub-formula drops below some clause limit and is solved by a brute-force solver, or the applied decisions satisfy all remaining clauses on their own. This algorithm has already been implemented and experimentally evaluated on a workstation cluster in order to demonstrate process migration techniques [1]. Due to the used platform and the different focus of research, the running times are far from the results obtained on our GPU.

2.1. Adapting the Procedure for Parallelism

During the parallelization of the sequential divide-and-conquer algorithm, two central elements of the procedure were altered, in order to better suit it to the underlying CUDA platform. First, the brute-force solver cutting off the recursion before it arrives at empty formulas was completely discarded. Second, the original decision heuristic to choose the next clause to be satisfied is replaced by a much simpler rule: Always choose the first clause. Both of these changes went through several iterations over the course of development, with the following findings leading to the current version of the procedure:

First, the brute-force solver breaks a pure SIMD paradigm, as most streaming processors are typically occupied with processing sub-formulas, and only a few would actually execute the brute-force solver. This alone would not be a big issue in itself, as it was already mentioned that different CUDA thread blocks may handle different tasks without seriously impacting the overall performance. However, in contrast to the default formula simplification, brute-force solving requires a certain amount of block-wide shared memory. Again, this fact alone is not an issue. In combination with the fact that most blocks do not execute the brute-force solver, however, this results in every block requiring the guarantee that this amount of shared memory is available, even though it mostly will not be used at all. On CUDA, this means the allocation of additional shared memory resources before the respective kernel even starts, which seriously impacts performance and is mostly not even required. Consequently the brute-force solver was discarded from the overall procedure, as its benefits did not outweigh its maintenance costs.

Second, the solver's decision heuristic. Simply choosing the first clause offers - at first - basically only a single advantage: It is *really* cheap. On the other side, there are a lot of arguments against it: Essentially every state-of-the-art solver employs complex decision heuristics and in doing so, benefits from partly huge performance improvements. However, the key word hereby is *complex*, which in our SIMD approach with hundreds of thousands of search paths running in parallel is just not feasible, as the required memory resources per formula are simply not available. Nevertheless, a multitude of different approaches was tested - among others, an additional heuristic kernel, employing a scoring system similar to the common VSIDS (Variable State Independent Decaying Sum) [18] heuristic - with the somewhat surprising result that it did not pay off at all to try and make good *local* decisions (i.e., per search path). Instead, a second heuristic was introduced, to complement the first-clause rule: Before actually solving an instance, it is first reorganized in order to sort its clauses by relevance. A clause is considered to be more “relevant” than another, if it contains a set of more often occurring variables. This order is preserved in every sub formula created during the following solving process, and we thus gain the property that the first clause of every formula being processed automatically denotes the globally recommended decision clause.

2.2. The Parallelization

The central approach of the parallelization it to avoid the consecutive execution of the recursive calls in the sequential version, by executing all (up to) three calls in parallel. Thus each input formula may spawn three new search paths, and consequently the amount of managed formulas triples during every step in the worst case. In order to deal with this, an additional swapping mechanism is introduced, which keeps the amount of formulas residing on the device on a near constant level. Swapping occurs after a batch of formulas has been processed by the device, and either transfers a formula surplus back to the host, or supplies the device with previously swapped out formulas in case too many of the newly created formulas feature contradictions and are discarded. The overall implementation may hence be designated a master-slave approach, as the slave/device iteratively processes formula batches which are continuously refilled by the master/host.

As this means that every step of the solver loop includes the transfer of quite some data between host and device, all data is being bit-encoded to reduce the resulting traffic costs. A single search path consists thereby of its (partial) variable assignment, and the actual formula resulting from applying this assignment to the original problem instance. The bit encoding allows one 32 bit integer to store a complete clause, and two bits per variable (while one bit is enough to encode *true*/*false* assignments, two bits allow for additional states such as *unknown* and *error*).

To access all this data, we employ a "read once, write once" strategy. Usually in CUDA programming, threads read their workload from global memory into shared memory, execute their function, and write the result back to global memory. The locality of our approach, however, allows for the circumvention of the shared memory entirely: Threads do not need to communicate with each other, and only work on a single clause at the same time. This clause is encoded within a single integer, and therefore is loaded once from global memory into a register. There it is being operated on as a temporary variable and immediately written back to global memory. Due to our development on the GT200 platform, we are able to benefit from relaxed rules for coalesced memory access, which allows for this strategy to be handled efficiently by the underlying CUDA hardware, and thus to actually improve the solving procedure.

Note that while single threads handle single clauses, a complete formula is not being processed by a single thread, but by a set of 32 threads, i.e., a warp. By issuing a single warp to handle a single formula, we can operate on the formula in parallel without any need for additional costly synchronization between the threads, since warps are guaranteed to be physically executed in parallel on the device. Therefore, our kernels operate on a CUDA grid comprised of thread blocks containing 192 threads, which split up into 2 • 3 warps. Three warps at a time access the same *input* formula and produce a unique *output* formula. Thus a single block reads two input formulas and employs six warps to produce six different output formulas.

In order actually start the solving process, an additional pre-process kernel is invoked. This kernel takes the initial problem instance and optimistically applies every assignment permutation of the x most occurring variables. The value of x is chosen to immediately obtain a completely filled formula batch, thus avoiding a partially idle startup phase on the device. This initial batch is then being fed to the actual solving mechanism - the *kernel pipeline*.

2.3. The Kernel Pipeline

The kernel pipeline denotes a series of kernels which are successively applied to the formula batch currently residing on the device. Each of these kernels performs a single operation of the overall solver loop. After being processed by the kernel pipeline once, a single step of the divide-and-conquer recursion has been applied to a formula batch containing n formulas. Furthermore, all thereby newly created $3 \cdot n$ formulas have been checked for contradictions and validity, and the next formula batch has been constructed from all remaining valid formulas.

More specifically, the kernel pipeline executes the following steps:

- *SAT Rules:* Triple the number of formulas within the batch by applying new assignments to each formula according to the rules of Algorithm 1.
- *1D Prefix Scan:* Scan all formulas for validity, in order to discard any conflicting search paths marked by the previous SAT kernel.
- *Copy:* Use the prefix scan result to condense all valid formulas into a new, compact formula batch.
- *2D Prefix Scan:* Scan all remaining formulas, in order to discard any nullified clauses.
- *Defragmentation:* Use the 2D prefix scan result to rearrange all remaining valid formulas in order to form continuous formula strings without nullified clauses mixed in between.
- *Swapping:* Adjust the size of the new batch by either trimming or reinforcing the amount of contained formulas. (Note that this is a host-side action and thus not actually performed by a CUDA kernel. It is included as a part of this list for the sake of convenience only.)

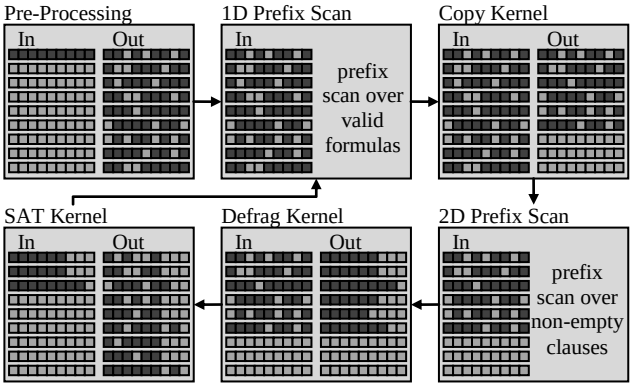

Figure 1. The Kernel Pipeline

A complete overview of the kernel pipeline is provided by Figure 1, where for each step the state of the active formula batch is shown before entering and after leaving the respective CUDA kernel. One row within the batch corresponds to a single formula and each formula consists in turn of several blocks representing the formula's various clauses. A filled block denotes a valid clause, while an empty block depicts either an invalid clause or a clause slot which is no longer in use. As can be seen, the initial formula is taken by the preprocess kernel, which generates the first actual formula batch. When creating new formulas, any involved thread may set a flag to notify its warp that the currently processed formula is invalid. These flags are then checked by the following kernel, which employs the CUDPP library [11] to run a parallel prefix scan over all the flags of the active formula batch. The result of this scan is subsequently used in the following kernel, which collects all valid formulas into a new formula batch. Such newly created formulas, however, are still littered with nullified clauses, resulting from the last decisions being applied. To pack these clauses tightly together in order to form uninterrupted formula strings again, CUDPP is employed once more to run a 2D prefix scan over all formulas and their respective clauses. The result of this scan is then used by a defragmentation kernel which compresses all valid formulas by reordering their clauses and thus regaining continuous formula sequences.

The final step consists of checking the actual size of the newly formed formula batch, and invoking the swapping mechanism if necessary: In case enough formulas were produced to overwhelm the hardware during the next expansion, some of those formulas are being swapped back to the host. In contrast, if most of the newly created formulas turned out to be invalid and thus had to be discarded, the host swaps previously stored formulas back to the device, in order to resupply the now lacking formula batch. Are no formulas left to swap back in, the search space has been exhausted and the formula is not satisfiable. Note that swapping is not explicitly depicted in Figure 1, as it is not performed by a CUDA kernel on the device, but by the host itself.

As can be seen, most steps of the kernel pipeline merely rearrange formula data to generate the solid formula batches being processed by the SAT kernel. This kernel is where most of the actual solver work is being done: Variable decisions are being applied, and new formulas are generated. It is executed right after the swapping mechanism has adjusted the new batch size, and will be explained in more detail, as it is the essential kernel of the overall solver:

As its first step, each warp determines its own workload, i.e., which of the formulas contained in the current batch the warp is supposed to process, which route of the core recursion it has to follow, and where the actual result should be stored. New formulas are then being generated by taking the first clause of the input formula and applying the assignments denoted by the core recursion. To this end, each warp employs its threads to iterate the clauses of its input formula and to work through the following set of solver rules:

- *Subsumption:* Discard any clause containing a decision literal.
- *Resolution:* Discard any occurrence of a negated decision literal in all clauses.
- *Conflicts:* In case any clause of the formula contains only a negated decision literal, mark the overall (output) formula as invalid.

Note that it is not always possible for every warp to create a new formula: If the decision clause contains less than three literals, the solver is unable to expand the respective input formula into three new output formulas. In this case, the respective warp acknowledges its missing purpose and shuts itself down. On the CUDA side, the streaming processor executing this warp is then immediately being tasked with the execution of one of the remaining warps by the CUDA scheduler. Accordingly, as long as the solver provides enough work for the device, the streaming processors can simply be flooded with work and the device will operate at around its full capacity. Thus, no additional load balancing mechanism is necessary, as all tasks are distributed to the available processors automatically be the running CUDA scheduler.

Once all warps are done, the resulting fragmented batch of clauses is then being fed back into the kernel pipeline. This loop continues until an empty formula is created - and the formula thus was satisfied - or all available search paths have been exhausted. In both cases, the program terminates and reports the obtained result.

As can be seen, the overall procedure is deterministic. It might not be known in what order the formulas of a batch will be processed, but all decisions being made are known beforehand and are not dependent on any random scheduling behavior. The only randomness that might occur (and has been confirmed in experiments) takes place when the solver produces two or more solutions in the same step. In this case, it cannot be guaranteed which warp will succeed in constituting its own solution as the solver's final one.

3. RESULTS

When presenting the outcome of implementing our massively parallel solver, two central aspects are considered: First, the results delivered by the actual solving of SAT instances, and second an evaluation of the algorithm's parallelization in general and on CUDA hardware.

The benchmark problems used to evaluate our solver were taken from the open Satisfiability Library [12]. As it was the primary goal to achieve a successful parallelization, we chose a set of comparatively small problems, in order to be able to quickly experiment with small scale modifications and parameter variations. The set consists of 100 random instances, all known to be satisfiable and containing 75 variables and 325 clauses - resulting in a clauses-to-variables ratio of 4.33, slightly above the known threshold for problem hardness.

Figure 2 presents the results of solving the complete input set on two different CUDA graphics cards, specifically a GeForce GTX 285 (blue/dotted line) and a GeForce GTX 260 (red/plain line). While the x-axis depicts the number of solved instances, the y-axis denotes the accumulated amount of time needed to solve this number of instances in seconds. Both cards utilize (among other things) a different number of parallel streaming processors (240 on the GTX 285 and 216 on the GTX 260), which - besides a more complex SLI setup - is basically the only way to check CUDA algorithms for processor scaling.

Figure 3 is laid out the same way, and depicts the results of solving the same instance set on the GTX 285 alone, while using three variations of the central solver parameter. This parameter denotes the amount of formulas handled concurrently, and is limited by the number of formulas the graphics card is able to store at the same time. Note that the fastest configuration is actually not the one using the most memory resources, as the continuous formula transfer from host to device eventually begins to slow down the overall solving process.

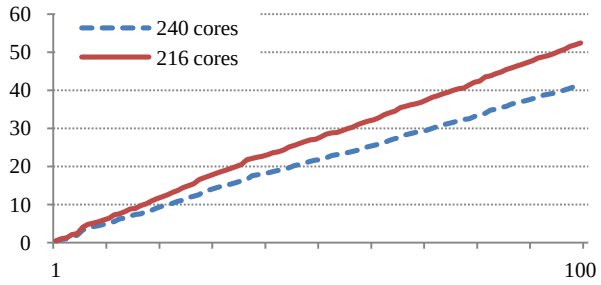

Figure 2. Solving the SATLIB 75-325 Set

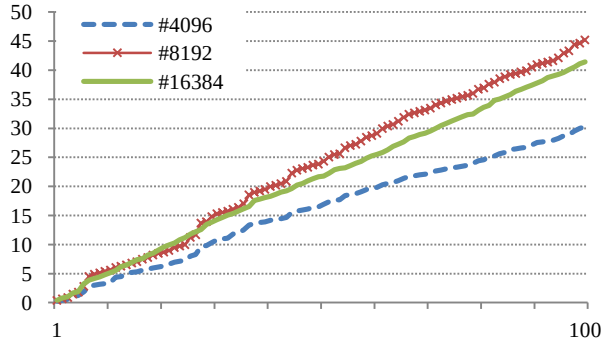

Figure 3. Adjusting the Batch Size

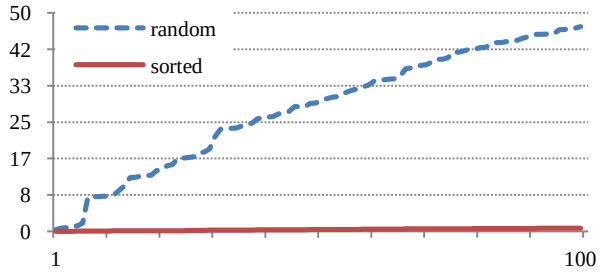

Figure 4. Decision Heuristic Impact

Figure 4 shows the impact of our two complementary decision heuristics (sorting by "relevance" and always choosing the first clause). Again the same set of 100 formulas is being solved on the GTX 285 card, once with the un-altered formulas (blue/dotted line), and once with all formula clauses sorted by relevance (red/plain line). Note that the y-axis of this figure denotes not seconds, but minutes.

Adhering to the SIMT paradigm, the parallelization itself depends on a massive number of threads flooding the parallel hardware with work, and thereby trying to never leave one of the processors idle and waiting for something to do. Furthermore, the parallelization is designed to break up into individual threads in the last possible moment (i.e., when it comes to actually iterating a formula's clauses), and thereby minimizing divergence within the executed warps. In order to evaluate how successful such goals are being accomplished, CUDA

provides a profiling tool which employs a wide range of counters to help evaluate the successful use of the utilized CUDA hardware. Table 1 shortly summarizes the overall useful activity of a streaming processor during the execution of our solver.

Table 1. Profiler Results

CUDA Kernel	*Occupancy*	*Instruction Throughput*
SAT Kernel	0.75	1.00
Copy Kernel	0.94	0.58
Defrag Kernel	0.94	0.91

3.1. Discussion

The main issue with our approach lies where sequential algorithms gain most of their performance: Collect various amounts of state data while solving, feed this data to a set of complex heuristics, and use the resulting advice to make smart decisions which lead to some ingenious route through a gigantic SAT search space. Our framework, however, does not allow for such costly mechanisms per followed path, as the required resources are not available on CUDA. While being a clear disadvantage, this also forced the development to primarily focus on actual parallelism to organize and speed up the solving process. Consequently, we had to re-think, and often discard, common sequential solving techniques, in order to fully take advantage of the parallel CUDA processing power, and exploit the highest level of thread parallelism possible. Accordingly, the hereby presented solver framework should not be viewed as a finished product, but rather as an initial stepping stone towards a competitive parallel SAT solver.

As such, there is no comparison chart with contemporary sequential solvers. They are built on a highly advanced DPLL framework and trump our solver by magnitudes. Although we were able to gain several performance boosts during the development of our framework, there is simply no competition worth speaking of. Furthermore, the most useful comparison with another solver would be a different massively parallel solver on a multicore platform, in order to determine the success of alternative solving approaches in this setting. However, to our knowledge, there are none such solvers currently available.

4. FUTURE WORK

In its current version, our solver framework offers three specific degrees of freedom where new heuristics may be applied, in order to improve our decision making: First, when swapping formulas (in or out), it has to be decided which ones are moved. Second, the process to determine the next decision clause may be further worked on, as it is the most relevant aspect of how the search is organized. Third, the order by which newly generated formulas are being organized when forming a new formula batch might be reworked, in order to complement the currently employed swapping strategy. While we already propose solutions for all of these - and were able to produce some interesting results - our implemented techniques are not comparable with the level of sophistication commonly found in state-of-the-art sequential solvers.

Besides new heuristics, it also might pay off to remodel some of the internal framework. Specifically, we have observed that our solver is able to process an impressive number of formulas - around 5.5 million per second on average on the GTX 260 - and at the same time (predictably) being slowed down by costly memory transfers. Accordingly, it might be beneficial to further reduce said transfer costs by not modeling a single search patch by partial assignment *and* the respective sub-formula, but instead only the partial assignment. In order to still be able to detect conflicts, formulas would then have to be constructed on demand. While this would drastically reduce the amount to be transferred per solver loop iteration, it would require the allocation of enough shared memory resources to store such temporarily generated formulas, and the additional computational overhead of actually handling them.

Furthermore, note that the solver might also very easily be expanded to run on more than just a single CUDA card: In order to be executed on *n* cards, the pre-processing kernel simply needs to generate an initial batch of *n* times the size a single card is able to handle. This batch is then split up, and distributed to all *n* available graphics cards.

4.1. Perspective for Massive parallelism

It is probably safe to say that parallel platforms will continue their successful rise. Therefore, it seems only a matter of time until state of the art solvers incorporate the use of massively parallel hardware, and routinely involve techniques depending on the advantages offered by such platforms. This paper does not present such a solver. We would like to think, however, that this paper presents some groundwork, from which such a solver may be built in the near future.

How this may actually proceed, however, is still open to debate. With the development of high end sequential solvers, we see the careful guiding of a *single* search path on one side of a spectrum, with massive parallelism and millions of concurrent - but largely unguided - searches residing at the opposite end. While solvers like ManySAT move from the single path end to the parallel side, we started at the purely parallel side and slowly developed onwards to the more guided side of the spectrum. It will

probably be interesting to see what kinds of solver emerge once these two paths meet in the middle, at what point of the spectrum the most successful solver will be found, and what part may be handled most effectively by actual parallelism within an overall solver framework.

Consequently, there may be two basic approaches when further developing parallel solvers: Either expanding parallelism in hybrid solvers - the common contemporary approach - or finding some way to employ known successful techniques from the sequential toolbox within a resource-scarce parallel environment.

5. CONCLUSION

In this paper, we presented a SIMT 3-SAT solver framework, which discards most common solver techniques introduced by sequential solvers, and instead solely relies on massive thread parallelism to solve problem instances. To share actual results, the solver was developed to run on available Nvidia CUDA hardware. The underlying algorithm, however, does not rely on any platform specific features, and thus may very straightforwardly be brought to any other SIMT platform as well. Our central goal was not to present the next high end SAT solver, but rather to determine what massive parallelism might actually be able to bring to the table of SAT solving. The solver offers various degrees of freedom to easily allow for further experimentation and expansion of the framework - all within a comfortable desktop environment, without the need of a large computer cluster setup to which the solver might be brought on a later stage.

REFERENCES

[1] O.Bonorden. "Load balancing in the bulk-synchronous-parallel setting using process migrations". In *Proc. 21st Int. Parallel and Distributed Processing (IPDPS)*, pages 1–9, 2007.

[2] W. Chrabakh and R. Wolski. "GrADSAT: A Parallel SAT Solver for the Grid". Technical Report 2003-05, UCSB, Mar. 2003.

[3] W. Chrabakh and R. Wolski. "GridSAT: A system for satisfiability problems using a computational grid". *Parallel Computing*, 32:660-687, 2006.

[4] S. A. Cook. "The complexity of theorem-proving procedures". In *Proc. 3rd ACM Symp. On Theory of Computing (STOC)*, pages 151-158, 1971.

[5] M. Davis, G. Logemann and D. Loveland. "A machine program for theorem proving". *Commun. ACM*, 5(7):394-397, 1962.

[6] M. Davis and H. Putnam. "A computing procedure for quantification theory". *J. ACM*, 7:201-215, 1960.

[7] N. Eén and N. Sörensson. "An extensible SAT-solver". In *Proc. SAT 2003*, pages 502-518, 2003.

[8] S. L. Forman and A. M. Segre. "NAGSAT: A randomized, complete, parallel solver for 3-SAT". In *Proc. SAT 2002*, pages 236-243, 2002.

[9] M. R. Garey and D. S. Johnson, COMPUTERS AND INTRACTIBILITY – A GUIDE TO THE THEORY OF NP-COMPLETENESS. Freeman, New York, 1979.

[10] Y. Hamadi and S. Jabbour. "ManySAT: A parallel SAT solver". *Journal on Satisfiability, Boolean Modelling and Computation*, 6:245-262, 2009.

[11] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. "CUDPP: Cuda data parallel primitives library". http://www.gpgpu.org/developer/cudpp/, 2008.

[12] H. H. Hoos and T. Stützle. "SATLIB: An online resource for research on SAT". In *Proc. SAT 2000*, pages 283-292, 2002.

[13] J. Hromkoviç. ALGORITHMICS FOR HARD PROBLEMS. Springer, 2001.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "NVIDIA Tesla: A unified graphics and computation architecture". *IEEE Micro*, 28(2):39-55, 2008.

[15] Y. S. Mahajan, Z. Fu, and S. Malik. "Zchaff2004: An efficient SAT solver". In *Proc. SAT 2004 (Selected Papers)*, pages 360-375, 2004.

[16] J. P. Marques-Silva and K. A. Sakallah. "GRASP: A search algorithm for propositional satisfiability". *IEEE Trans. Comput.*, 48:506-521, 1999.

[17] B. Monien and E. Speckenmeyer. "Solving Satisfiability in less than 2^n steps". *Discrete Applied Mathematics*, 10:287-295, 1985.

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: Engineering an efficient SAT solver". *In Proc. 38th Design Automation Conference (DAC)*, pages 530-535, 2001.

[19] NVIDIA, "Next generation CUDA compute architecture, code named Fermi". http://www.nvidia.com/object/fermi architecture.html, 2009.

[20] H. Zhang. "SATO: An efficient propositional prover". In *Proc. Int. Conf. on Automated Deduction (CADE-97)*, pages 272-275, 1997.

[21] 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.