PARALLEL SAT-SOLVING WITH OPENCL

Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, Joost Vennekens
EAVISE, Lessius Mechelen – Campus De Nayer, Belgium
sander.beckers@mechelen.lessius.eu

ABSTRACT

In the last few decades there have been substantial improvements in approaches for solving the Boolean satisfiability problem. Many of these improvements consisted in elaborating on existing algorithms. On the side of the complete solvers this led to more efficient branching heuristics and the use of watched literals for unit propagation; incomplete solvers on the other hand have benefited from using random walks and from integrating evolutionary algorithms into the search process. Another line of research has combined these developments by creating hybrid solvers. More recently, the rapid development of parallel architectures has led to improvements in the implementation of algorithms as well, both on the side of complete as on the side of incomplete solvers. Up until now this work has been limited mostly to grids or multicore CPU's. Recent advances such as the OpenCL framework however, make it possible to also exploit the processing power of today's GPU's, and do so together with the CPU. Our goal is to take advantage of this, by implementing a parallel version of a hybrid algorithm that has been proven remarkably succesful in the class of locally inconsistent SAT problems. By parallelizing local search, our solver will be competitive on consistent problems as well. The workload of the hybrid algorithm will be split between the GPU and CPU, the former performing a massively parallel local search and the latter an adapted version of MiniSAT, a popular complete solver. We present here the results for a first simplified implementation, and for simulations of the final solver.

KEYWORDS

SAT, OpenCL, GPGPU.

1. THE SAT-PROBLEM

The Boolean satisfiability problem consists in deciding whether a given propositional formula in conjunctive normal form can be satisfied by a suitable choice of assignments to the propositional variables. For example, consider the following problem:

You are throwing a dinner party. Your wife wants you to invite either George or to exclude Jones. You want to invite at least one of Jones and Tom. But Tom can't stand Jones, so you can't invite both. Is there any way for you to succesfully organize this party?

We can express this problem as a satisfiability problem through the following formula:

$$(G \vee \neg J) \wedge (J \vee T) \wedge (\neg T \vee \neg G)$$

where G, J and T are Boolean variables which represent the invitation of, respectively, George, Jones and Tom. This formula is in conjunctive normal form, meaning that it consists of a conjunction of so-called clauses, which are disjunctions of literals. Out of the eight possible assignments, only two satisfy the formula, namely either you assign true to G and J and false to T, or you assign false to G and J and true to T . So either you invite George and Jones or you only invite Tom.

The satisfiability problem historically played a major role in complexity theory, since it was the first problem to be proven NP-complete (Cook S.A., 1971). (Unless $P=NP$, this means that we can never find a worst-case polynomial algorithm that solves the SAT-problem. However it turns out that in practice many problems can be solved efficiently.) The reason why so much research has gone into finding efficient

algorithms is that nowadays the SAT problem is widely used to model combinatorial problems in many areas, such as AI planning, software model checking and electronic design automation, to name but a few.

There are basically two types of solvers for the SAT problem, namely complete solvers and incomplete solvers. Complete solvers are guaranteed to decide for every SAT instance whether or not it is satisfiable, given enough time, whereas incomplete solvers may stop before finding a solution even if one exists. The benefit of the latter is that they are in general a lot faster on satisfiable instances than complete solvers, but have the obvious disadvantage that they are incomplete. To repair this defect and have the best of both worlds, hybrid solvers were developed. These combine both type of solvers in a way that tries to retain as much as possible of their respective advantages.

1.1 Complete solvers

The backbone of all complete solvers that have been developed over the last few decades remains the DPLL algorithm that was discovered in the 60's (Davis, M. et al., 1960). The algorithm is a recursive, depth-first enumeration of all possible assignments in the model space, which can be seen as a binary tree. The progress in this area was mainly due to intelligently optimizing the different parts of the algorithm, such as the heuristics used to choose the next branch in the search tree, and the data structures used for the propagation of unit literals. An additional improvement came from expanding the original problem by adding new clauses that have been learned from conflicting assignments.

For our research the branching heuristic will be most significant. It determines in which order the search tree is traversed, and choosing a relevant heuristic can therefore drastically reduce the number of recursive calls to be made in order to find a solution. A lot of modern SAT solvers use the Variable State Independent Decaying Sum (VSIDS) heuristic (Moskewicz, M.W. et al., 2001), which assigns a score to each literal according to its activity in the search process.

1.2 Incomplete solvers

A typical incomplete solver as described in (Selman, B. et al., 1992) uses a greedy local search algorithm, where the variables are initiated with a random assignment, and at each iteration a single variable is "flipped" to its opposite value. The variable to be flipped is chosen such that the number of satisfied clauses is maximized. This continues until either a solution has been found, or a fixed limit of flips has been performed. In the latter case, the process is repeated, for a set number of tries. To avoid ending up in local minima, the greedy approach is usually combined with a random walk, so that with a certain probability a variable that appears in an unsatisfied clause is chosen at random instead of the greedy choice (Selman, B. et al., 1993). This simple algorithm performs well on a wide set of instances.

1.3 Hybrid solvers

As noted, the disadvantage of incomplete solvers is that one cannot be sure whether the instance is indeed unsatisfiable if the solver fails to find a solution. On the other hand complete solvers and their activity based heuristics have great difficulty in solving really large and hard unsatisfiable instances. For these cases the VSIDS heuristic fails in efficiently guiding the search process. It turns out that, in a hybrid approach, both of these defects taken together cancel each other out. Mazure et al. (1997, 1998) use a local search algorithm to provide a branching heuristic for DPLL that dramatically decreases the runtime of hard unsatisfiable random instances, especially those that have a locally inconsistent kernel.

First, their TWSAT extends the local search algorithm by adding a Tabu list, so that recently flipped variables may not be flipped, preventing the algorithm from getting stuck in local minima and from moving back and forth between a small set of assignments. During the local search, the following trace is recorded: for each literal, it is counted how many times it appears in a falsified clause, taking a flip as a measure of time. Literals that have a high score are more likely to belong to the inconsistent part of the instance than those with low scores. Indeed Mazure et al. discovered that using this literal score as a branching heuristic for a standard DPLL algorithm, whereby TWSAT is called every time the DPLL algorithm needs to choose another literal and the literal with the highest score is selected, gives excellent results on locally inconsistent problems.

2. FROM HYBRID TO PARALLEL

With the increase of CPU speeds coming to a halt in the last few years, it becomes more and more clear that progress in purely sequential SAT-solving is reaching its end. Luckily parallel hardware is becoming ever more widespread and GPU's can be found in standard PC's. Thanks to the creation of new libraries, such as OpenCL, it is becoming possible to perform platform-independent GPGPU-computing. The goal of our research is to exploit these new possibilities for implementing parallel versions of SAT-solvers, where our current focus lies on parallelizing the Tabu technique described in the previous section. Most notably the recent advent of the OpenCL programming environment provides us with the tools for writing programs that combine CPU and GPU resources to suit one's needs, which fits in nicely with the hybrid character of the Tabu approach. The CPU is used to run a version of MiniSAT (Sörensson, 2003), a succesful and popular DPLL variant which normally uses VSIDS as a heuristic.

We adapt MiniSAT so that it no longer uses VSIDS, but a form of the TWSAT heuristic instead. However, rather than sequentially calling TWSAT every time MiniSAT needs to choose a literal, so that MiniSAT has to wait for TWSAT to finish, we run TWSAT in parallel on the GPU at the same time as MiniSAT. Every now and then the CPU informs TWSAT which variables have been assigned already at that stage, so that future tries of TWSAT can run on the partial problem determined by the current subspace in which MiniSAT is looking. Likewise, every now and then all literal scores obtained from the local search trace are sent to the CPU, which keeps an ordered list of literals based on this score.

However, we cannot expect large gains from simply splitting the hybrid algorithm so that both the complete and incomplete part run in parallel, since it turns out that the incomplete part in most cases takes up more than 95% of the time. Indeed, the main reason why the hybrid algorithm is outperformed by traditional complete solvers on general unsatisfiable instances is that calculating the TWSAT heuristic is a lot more time-consuming than its competitors. The primary reason to parallelize the two algorithms is because this allows the local search part itself to be massively parallelized on the GPU. This is done by running in parallel a large number of threads (depending on the GPU this can go up to 1600) of the TWSAT algorithm, each working independently from the other and each starting off with a different initial assignment. The TWSAT score for a literal is then taken to be the average of the scores given by each thread. The underlying motivation for such an approach is that we expect it to significantly reduce the time needed to calculate a suitable branching heuristic by dividing the workload. The result would be an algorithm that is extremely fast on large and hard unsatisfiable instances, as well as on other unsatisfiable instances, and performs good on most consistent instances. Figure 1 illustrates our approach.

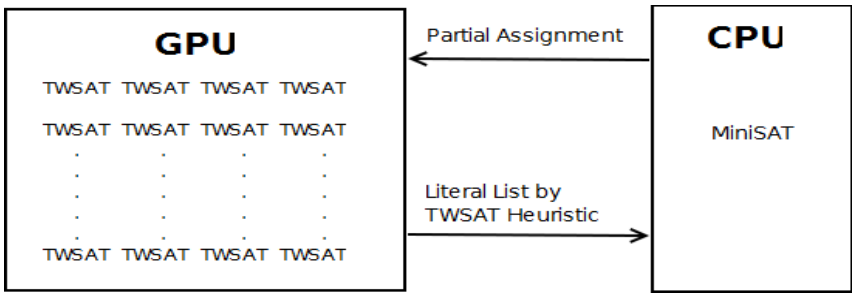

Figure 1. Our parallel hybrid SAT-solver.

3. RESULTS

At this stage of our research two forms of output can be presented that form the first steps towards an OpenCL implementation of a parallel TWSAT+MiniSAT solver. Firstly we have ported an existing massively parallel local search solver from NVIDIA's CUDA system (Wang, Y., 2010) to OpenCL (section 3.1). The framework of this version will form the basis of our parallel local search on the GPU. It will be

improved upon by optimizing memory use and data-transfer, and by replacing the current local search algorithm with TWSAT. Secondly we have simulated the output of the final implementation (section 3.2), in order to assess its strengths and weaknesses. This information will be helpful in deciding between different alternatives for developing our solver. We will now discuss these results.

3.1 Porting from CUDA to OpenCL

CUDA is a C-based programming language and environment for General Purpose GPU-computing created by NVIDIA. Its syntax and use are quite similar to regular C, except of course that special attention has to be paid with regard to synchronization for code that will be executed in parallel. Since CUDA was designed to run solely on NVIDIA hardware, a large part of the optimalization has been automated. OpenCL on the other hand is a recent open standard that was developed by the Khronos group[1], with the intention of providing cross-platform portability. This allows a tight integration of CPU and GPU threads, which suits the needs of our approach.

We have tested the solver on 100 random 3-SAT instances from the DIMACS library[2], where we took the average time of 4 runs. The results are shown in Figure 2. OpenCL contains a wide range of functions that allow the programmer to adapt the method of compilation to the platform at hand. Our current implementation of the parallel local search algorithm has not yet made use of these capabilities to improve the performance, which explains why it is still slower than the original CUDA version. As this is work in progress, the local search algorithm used in this version is still of a rather simple kind, to be used as a stepping stone to the final TWSAT procedure. Therefore comparisons with highly developed CPU-solvers would serve little purpose at this point.

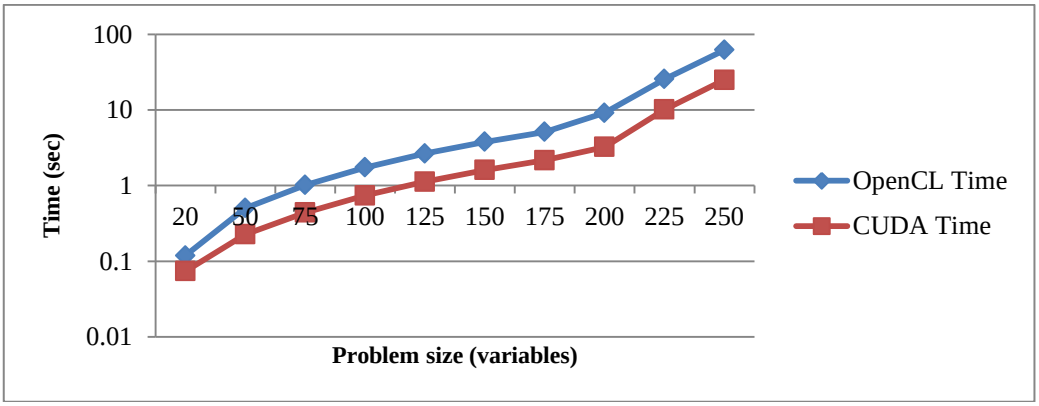

Figure 2. OpenCL vs Cuda solver

3.2 Simulation of parallel TWSAT-solver

In order to assess the viability of a fully parallel TWSAT+DPLL solver, we ran some tests that simulate what would happen if the TWSAT part would be parallelized into 20, 60 and 100 threads respectively. To this end we introduced some changes to the SUN-solver from (Mazure, B. et al. 1998), which is an implementation of the hybrid TWSAT + DPLL algorithm. More precisely, at each point where DPLL needs to choose a new literal, instead of running a single instance of TWSAT for N number of flips (N depending on the number of variables) we ran several independent threads of TWSAT sequentially. To estimate the time it would take a parallel version we added the purely DPLL time to the average time of all these threads, which is a fair assumption since each instance performs the same amount of flips. The original solver already performs extremely well on locally inconsistent cases - often taking less than a second, see (Mazure, B. et al., 1998) for

[1] http://www.khronos.org/opencl/
[2] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

results - so we chose to focus on random instances from the DIMACS library. In this way we can work towards a more general solver that performs well on all classes of problems.

During the tests we observed that simply adding more threads did not improve the TWSAT branching heuristic, a result that can be explained by the fact that the number of flips in the original version was already fine-tuned to obtain a satisfactory heuristic. Further we noticed that the majority of the time was spent on the TWSAT solver, and that the DPLL time was often negligible. Therefore instead of looking for an improved heuristic, a more promising approach was to reduce the time needed to obtain it. To do so we reduced the number of flips by 90%, which should be compensated by the increase of the number of threads. This method proved succesful. In fact, for the unsatisfiable instances, using only 20 threads turned out to result in a speed-up of a factor 3.5 for instances with 200 variables[3]. Further increasing the number of threads added little effect, and decreasing the number of flips even more proved dramatic for the usefulness of the TWSAT trace. The satisfiable instances show a slightly larger speed-up of factor 4, however this value holds for instances with 250 variables rather than 200. In this case there is a significant improvement by increasing the number of threads beyond 20. The results are shown in Figures 3 and 4, where in both cases the number of flips for multiple threads was 10% of the number of flips for 1 thread.

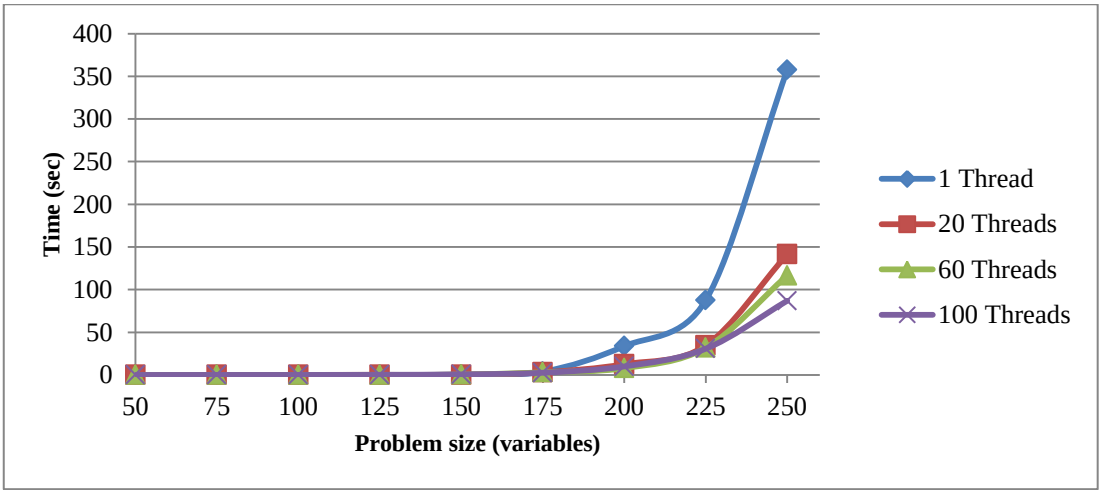

Figure 3. Satisfiable instances

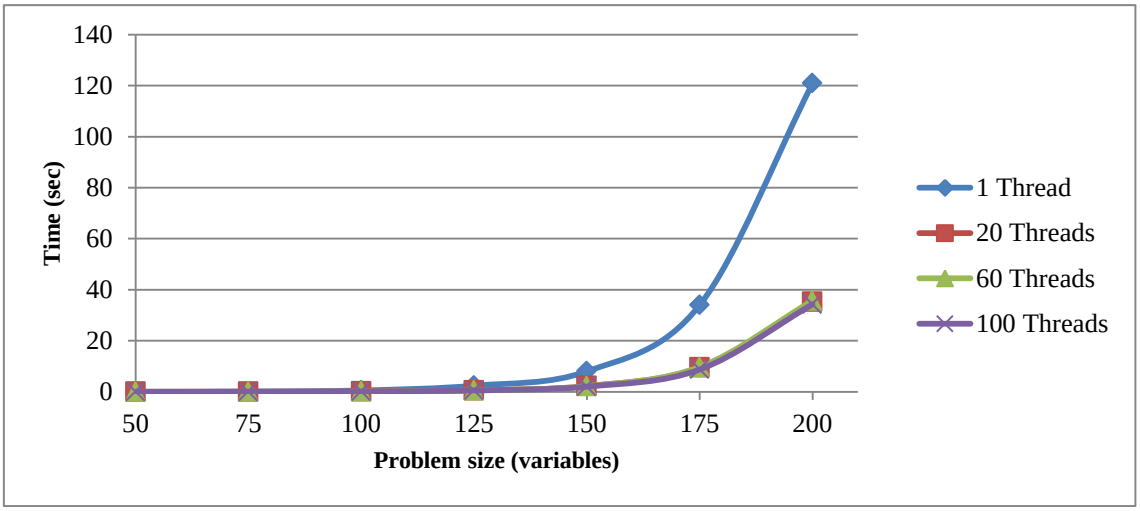

Figure 4. Unsatisfiable instances

[3] The clauses/variables ratio is 4.26 for all instances, meaning that they lie in the hardest region.

4. CONCLUSION & FUTURE WORK

We are investigating the use of the OpenCL parallel programming environment to implement novel strategies for solving the SAT-problem. At this point we have implemented a massively parallel solver on the GPU, and have developed an approach for enhancing the solver by adding a complete solver that will run on the CPU. Our next goal is to implement the hybrid TWSAT+MiniSAT solver with OpenCL, using the existing parallel local search implementation as a starting point. Secondly we will focus on optimizing, since it must be possible to perform at least equally well as the CUDA version. With the solver working, we will look at a larger group of test cases and compare with other solvers. Obviously the TWSAT heuristic has a limited scope, so in a final phase other heuristics will be constructed and implemented as well.

REFERENCES

Cook, S.A. 1971. The complexity of theorem proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation*, 151-158.

Davis M.; Putnam H. 1960. A Computing Procedure for Quantification Theory. In *Journal of the ACM*, vol. 7, 201-215.

Ferris, B.; Froehlich, J. 2004. WalkSAT as an Informed Heuristic to DPLL in SAT Solving. In http://www.cs.washington.edu/homes/bdferris/papers/WalkSAT-DPLL.pdf

Hamadi, Y.; Sais, L. 2009. ManySAT: a parallel SAT solver. In *Journal on Satisfiability, Boolean Modelling and Computation* (JSAT).

Mazure, B. et al. 1997. Tabu search for sat. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (AAAI '97), 281-285.

Mazure, B. et al. 1998. Boosting complete techniques thanks to local search methods. In *Annals of Mathematics and Artificial Intelligence*, 22:319-331.

Moskewicz, M.W. et al. 2001. Chaff: Engineering an Efficient SAT solver. In *Proceedings of the Design Automation Conference*, 530-535.

Selman, B. et al. 1992. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (AAAI '92), 440-446.

Selman, B. et al. 1993. Local Search Strategies for Satisfiability Testing. In *Working notes of the DIMACS Workshop on Maximum Clique, Graph Coloring, and Satisfiability*.

Sörensson, N; Eén, N. 2004. An Extensible SAT-solver. In *Lecture Notes in Computer Science*, vol. 2912, 333-336.

Wang, Y. 2010. NVIDIA CUDA Architecture-based Parallel Incomplete SAT solver. Master Project Final Report, Faculty of Rochester Institute of Technology.