Solving Satisfiability with a Novel CPU/GPU Hybrid Solution

Cas Craven, Bhargavi Narayanasetty, Dan Zhang
Department of Electrical & Computer Engineering
University of Texas, Austin, TX 78705
{dcraven, bhargavi, dan.zhang}@mail.utexas.edu

Abstract — **Boolean satisfiability is one of the most researched algorithms, finding uses in myriad fields ranging from artificial intelligence to formal verification. Most advances in the field have focused on improving the original Davis-Putnam-Logemann-Loveland (DPLL) algorithm. However, these optimizations were designed with single-threaded code in mind and many are not easily parallelizable. While parallelizing the original DPLL algorithm has been met with great success, achieving linear speedup, attempts to parallelize a modern DPLL algorithm with many advanced optimizations resulted in an overall slowdown in performance. We propose a hybrid solution utilizing NVDIA's CUDA platform, which runs on massively parallel graphics processors, as well as the system CPU, which offers high single-threaded code performance. Our preliminary testing demonstrates the potential for our novel solution, generating a 2.5x speedup over a standard CPU-only DPLL implementation.**

I. INTRODUCTION

The Boolean satisfiability problem (SAT) appears in many fields, including formal validation, artificial intelligence (AI), automatic test pattern generation (ATPG), timing analysis, delay fault testing, and logic verification. Due to the ubiquitious nature of SAT, combined with the fact that SAT was the first algorithm proven to be NP-Complete [1], considerable research effort has been spent designing efficient SAT algorithms. There are several conferences dedicated to the SAT algorithm and even a yearly competition. While many algorithms have been proposed, most are based on the original Davis-Putnam-Logemann-Loveland (DPLL) algorithm [2].

Several major improvements to the algorithm have been proposed since then. GRASP introduced an efficient method for *clause learning* [3], in which new clauses are appended to the original problem after discovering a conflict to avoid reaching the same conflict again. CHAFF [4] implements a highly efficient *Boolean constant propagation* algorithm, which identifies any variable assignments required by the current variable state to satisfy the entire equation. Tinisat [5] explores the effect of *branch restarting,* in which the DPLL algorithm is restarted on different branches of the tree to avoid traveling down a local maximum in conjunction with clause-learning.

While computer architects have moved toward exploiting parallelism instead of pushing the limits of single-threaded computing, not much work has been performed in multithreading the code. Current popular SAT solvers are all single-threaded implementations. Part of the issue is that the DPLL algorithm improvements are all single-threaded in nature and difficult to parallelize. While researchers have been able to demonstrate linear or even super-linear speedups when implementing the basic DPLL algorithm [6], attempts to parallelize a modern algorithm variant were met with failure [7]. In addition, all parallel work has been performed on networked computers via message passing or on SMP systems. GPUs are a low-cost and widely available platform that contains hundreds of simple cores, reaching more than 1 teraflop of theoretical compute power [8]. However, no work has been performed in porting DPLL to GPUs.

We propose a hybrid approach involving a CPU for running the single-threaded path selection and other optimizations while a GPU performs the heavy computation. We theorize that such a platform would be able to incorporate many of the latest single-threaded DPLL optimizations while gaining the compute throughput advantages of a GPU.

The rest of the paper is organized as follows. Firstly, we provide a brief background on SAT, the DPLL algorithm, and the CUDA GPU platform. Secondly, we describe potential partitioning of the algorithm on the CPU and GPU. Finally, we will provide initial performance results and analyze the resulting data.

II. BACKGROUND

A. The Boolean Satisfiability Problem

Boolean satisfiability is the problem of determining if there exists an assignment to a Boolean formula's variables such that the final result is true. If such an assignment exists, the formula is said to be *satisfiable*. Else, the formula is *unsatisfiable*. The formula is provided in *conjunctive normal form* (CNF), also known as a product of sums. This is not a limitation since any Boolean formula can be converted into an equivalent CNF formula in polynomial time [9]. The variables or negated variables in the function are known as *literals*. Literals are grouped into clauses through the use of a logical OR. Literals can be found any clause. Clauses are logically ANDed together to form the full CNF equation. For example, consider the following equation:

$$F = (x_1 + \overline{x_2} + x_3)(x_3 + \overline{x_4} + x_5)(\overline{x_1} + x_6)$$

This CNF formula contains three clauses: $x_1 + \overline{x_2} + x_3$, $x_3 + \overline{x_4} + x_5$, and $\overline{x_1} + x_6$. For an equation to be satisfiable, it should satisfy all of its clauses. An example solution would be: $x_3 = 1$, $x_1 = 0$, and all other variables are Don't Care. This particular SAT problem falls under the 3-SAT category, in which all clauses have at most 3 variables. 3-SAT is very popular in EDA and formal verification since all logic circuits can be represented in this form.

B. The DPLL Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm forms the basis of almost all 3-SAT solvers. The algorithm assembles the variables in a certain order and then incrementally assigns a value to each. As long as the resulting partial assignment doesn't falsify the entire formula (recall that since all clauses are ANDed together, if any clause fails then the entire equation fails), the algorithm continues to choose variables and assign values. If a clause fails, then the algorithm assigns the opposite value to the last chosen variable and checks again. If this falsifies another clause, then the algorithm knows that this entire branch is false. Then, the algorithm backtracks to another previously chosen variable and assigns it to be the opposite value. This search process resembles a depth-first search on a tree data structure. The algorithm continues this process until all variables are assigned and the formula is satisfied (SAT), or until all possible assignments have been checked and the equation is determined to be unsatisfiable (UNSAT). Since DPLL backtracks as soon as a branch is determined to be UNSAT, it won't necessarily visit all 2^n possible combinations.

DPLL incorporates an optimization technique called Boolean constraint propagation (BCP). In BCP, the algorithm searches through the clauses to see if the current partial variable assignment forces any other variables to must evaluate to true or false in order to satisfy the clause and thus the entire equation. This extra variable assignment can cause a cascade in which other variables must be set to a certain value.

The data structures in DPLL are fairly simple. The current variable assignment is represented as a large data array, with the 0th entry reserved. The index of the array represents the variable. Variables start with x1 and continue to x*n*, where *n* is the total number of variables. For example, the current value of x1 resides in the 1st entry in the array, the value of x2 is located in the 2nd entry, etc. The value can be one of three things: positive, negated, or undetermined.

SAT solvers usually take input files in the *DIMACS* CNF format, such as the following example:

```
c Here is a comment.
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

The format is very simple. Lines starting with c are comments. The first non-comment line of the file is of the format: *p cnf numLiterals numClauses*. Finally, every clause in the equation is listed, with the number representing the variable. Since zero can't be negated, we do not use 0 for a variable name. Each literal ends with a space and the number 0.

To run the basic DPLL algorithm, the processor simply performs a lookup in the variable assignment array using the variable number as the index.

The DPLL algorithm is very compute-intensive. Data sets are usually very small, on the order of several megabytes. In addition, while the basic DPLL algorithm can be easily be parallelized by partitioning the tree, many optimizations are single-threaded in nature (such as BCP) and cannot be easily parallelized.

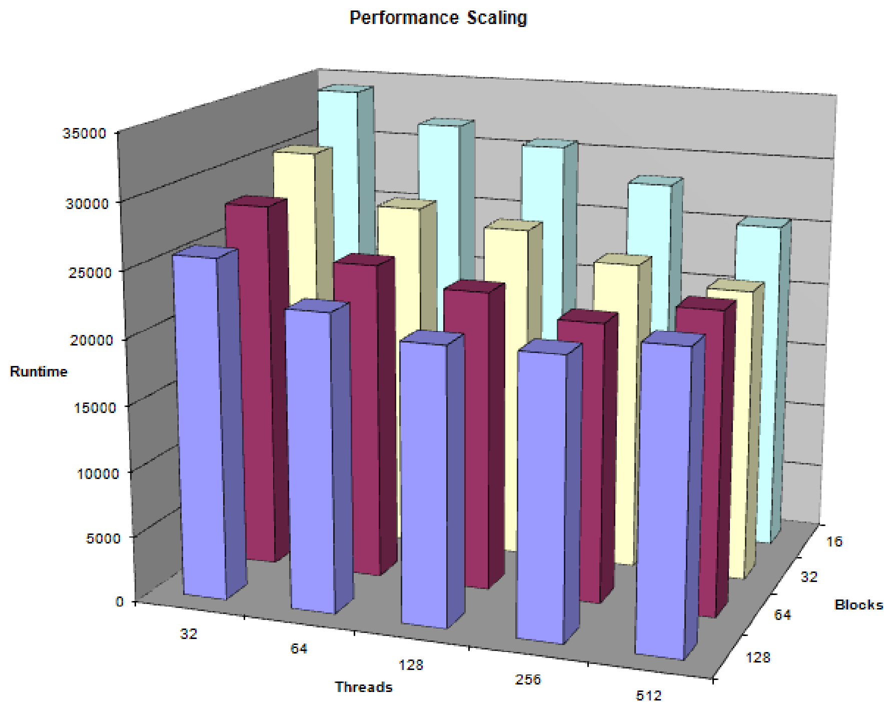

Figure 1. Hybrid SAT algorithm runtime, striding across the number of blocks and threads per block within CUDA

C. The CUDA GPU Platform

CUDA is an infrastructure from NVIDIA Corporation that provides software extensions to run general-purpose C code on their DirectX 10 graphics processors (GPUs). The architecture is organized in a hierarchical manner. On the NVIDIA GeForce 9800, there are 128 simple in-order stream processors which are organized in a SIMT (Single Instruction Multiple Thread) fashion. In other words, these 128 cores are grouped into 16 processing cores, each with 8 stream processors. Within each processing core, the same instruction stream is sent to each of the stream processors. To solve issues with execution divergence (for example, in an if…else loop), the stream processors use predication. The processing cores are all connected to a large 256MB or 512MB global memory implemented in GDDR3. To improve effective bandwidth and latency, each processing core also has a 16KB scratchpad that is shared between the stream processors.

Instead of optimizing for latency like CPUs, GPUs are optimized for throughput. Thus, the CPUs can switch threads at an extremely fine granularity to hide latency. At least four threads are necessary per stream processor in order to fully hide dependency and register file/shared memory access latencies and obtain maximal performance.

III. METHODOLOGY

Due to limitations of time, we only implemented the basic DPLL algorithm without Boolean constraint propagation. Since the goal of our project is to demonstrate the effectiveness of our CPU/GPU hybrid approach, we feel that our results satisfy the requirements. Obviously, since our implementation scales very poorly with larger datasets, we are only able to provide results for smaller inputs. However, our presented optimizations are used effectively in different algorithms and should definitely scale to larger sized inputs.

We tested several inputs that were automatically generated through a script. Some of the equations were SAT and some were UNSAT. We found that since our implemented optimizations were simple and work for everything, the results were fairly homogenous across different datasets. Thus, in the interest of time and space, we only present the results for a single testcase that has a runtime of a few minutes for the CPU-only implementation.

IV. CPU/GPU ALGORITHM

A. Initial Implementation

To take advantage of the CPU's single-threaded prowess and the GPU's ability to run highly parallel code, we designed a simple algorithm that tightly coupled the CPU together with the GPU. At the start of the program, all of the clauses are

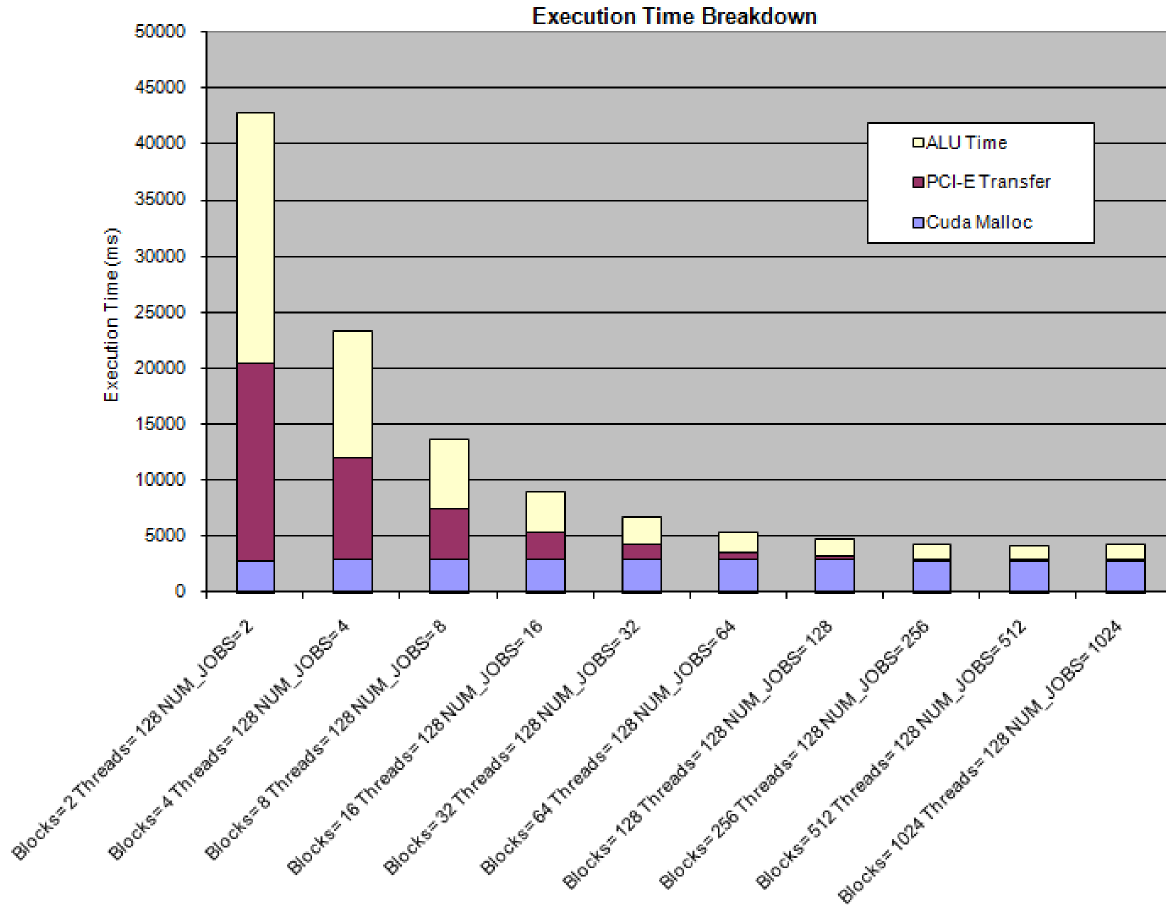

Figure 2. Breakdown of execution time into calculation, GPU/CPU transfer, and initial overhead for various job sizes

placed into the GPU global memory. During each iteration of the DPLL algorithm, the CPU will provide the GPU with the variable assignment. Then, while the GPU is processing the variable assignments, the CPU will calculate the next variable assignments for the GPU to consume. The GPU partitions the work as follows: clauses are placed within global memory. Each thread reads a few threads and accesses the variable assignment array, which is placed within shared memory. After each thread is done with their calculation, a parallel merge is performed and the result is sent to the CPU at the end of the kernel. Figure 1 shows the performance of our algorithm across a number of block and thread configurations. The algorithm clearly prefers a high number of threads to hide the latency of an access to global memory.

However, many problems became apparent during the implementation of our algorithm. For example, the CUDA kernel call is indeed non-blocking such that we can run code on the CPU in parallel with the code on the GPU, but the GPU cannot transfer data to the CPU until after the kernel call is over. Thus, we are not able to hide the latency of GPU to CPU communication, which is quite substantial since it's over the PCI-E bus. Even when we started sending only necessary data (a difference of ~2KB vs. 1 byte), the amount of time didn't change since it is completely latency-bound.

B. Parallel Depth First Search Optimization

To reduce the number of GPU to/from CPU transactions, thus reducing the bottleneck, we decided to parallelize the DPLL algorithm in a different dimension. Previously, we only parallelized the literal calculations. Our new optimization breaks down the DPLL tree into several regions, in which each thread block processes a subsection of the tree. This breakdown results in a minimal increase in the memory usage: now, *n* variable assignment arrays are stored in global memory within CUDA instead of only one. However, these arrays are fairly small and the datasets fit global memory with room to spare. In addition, no additional space in shared memory is used.

Keep in mind that the GPU is still only used to

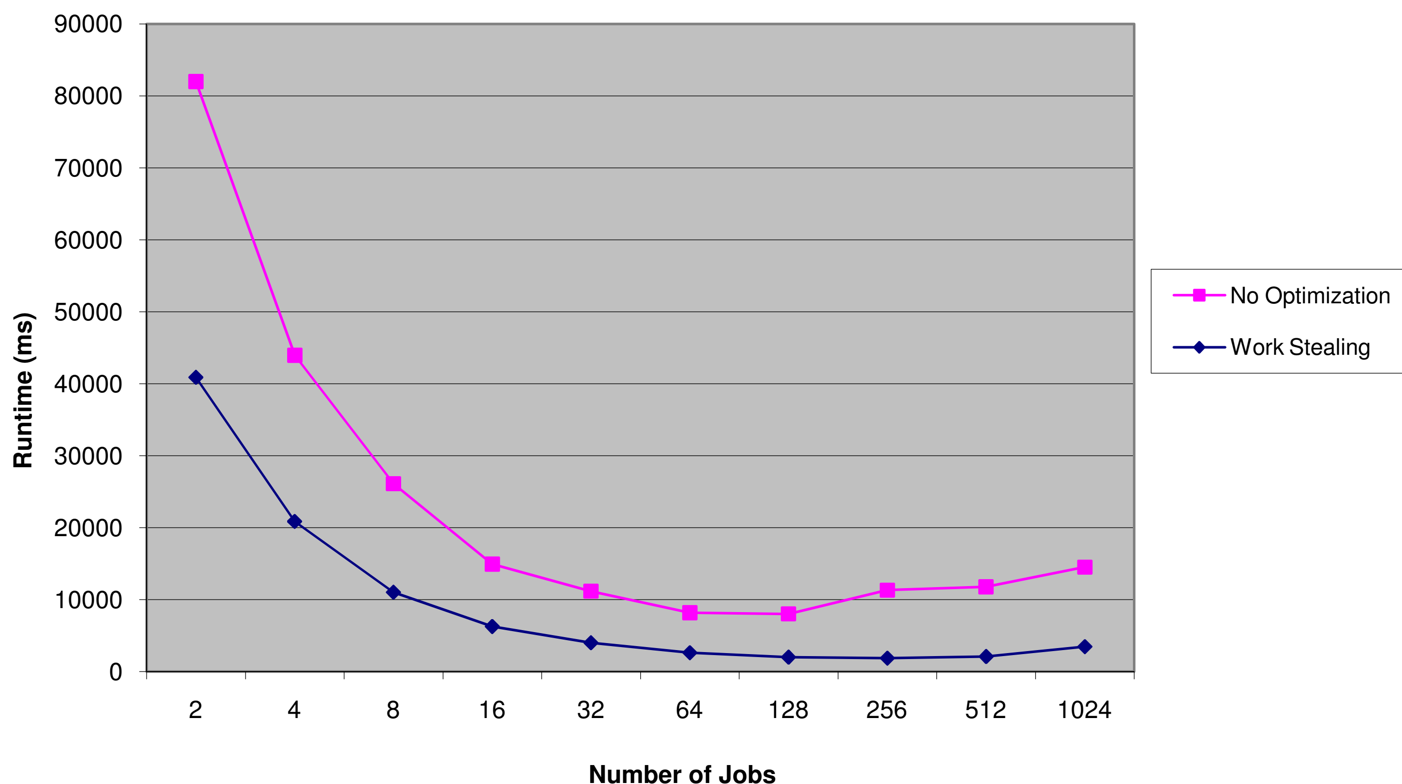

Figure 3. Increasing the number of jobs with and without work stealing

process the literals. The goal of this optimization is to aggregate GPU to CPU communication and also provide more work for the GPU to consume. Parallelizing the control goes against the goal of this project, which is to enable the CPU to perform the control work while giving the easily parallelizable jobs to the GPU.

As shown in Figure 2, this optimization provides a large amount of speedup as expected. Communication overhead between the CPU and GPU was decreased significantly. However, we noticed that there was a large variance in the individual job runtimes. There would frequently be a case in which multiple jobs finish quickly, with only a few jobs left remaining. Since the jobs are each mapped to a few processing units, we were clearly underutilizing the resources within the GPU. This was because DPLL could sometimes determine that a branch in the tree was UNSAT very early in the process. We therefore decided to implement another optimization to balance the amount of work per thread.

C. Work Stealing

Work stealing is a common method used to balance the amount of work performed per thread. The key concept is that jobs that finish early take a portion of the work left from jobs that have yet to be completed.

Our work stealing algorithm works as follows: when a job detects that its path is UNSAT, it runs a heuristic to determine the job that's furthest from being done. Our chosen heuristic is simple: since our DPLL algorithm searches the left-hand side (represented as a 0) of the tree first and only searches the right hand side (represented as a 1) as it backtracks, we look for the job that has a 0 closest to the root of the node. Then, we mark that section of the tree as having already been explored to prevent others from repeating the same work.

As shown in Figure 3, our algorithm works very efficiently. Since the CPU sends the entire variable assignment vector to the GPU, this work stealing algorithm demonstrates the potential of our hybrid CPU/GPU approach. Analysis shows that there is absolutely zero overhead as a result of work stealing on the GPU-side: the work-stealing algorithm is run on the CPU-side only. Our tests show that all threads remain perfectly balanced, some threads stealing work during back-to-back cycles.

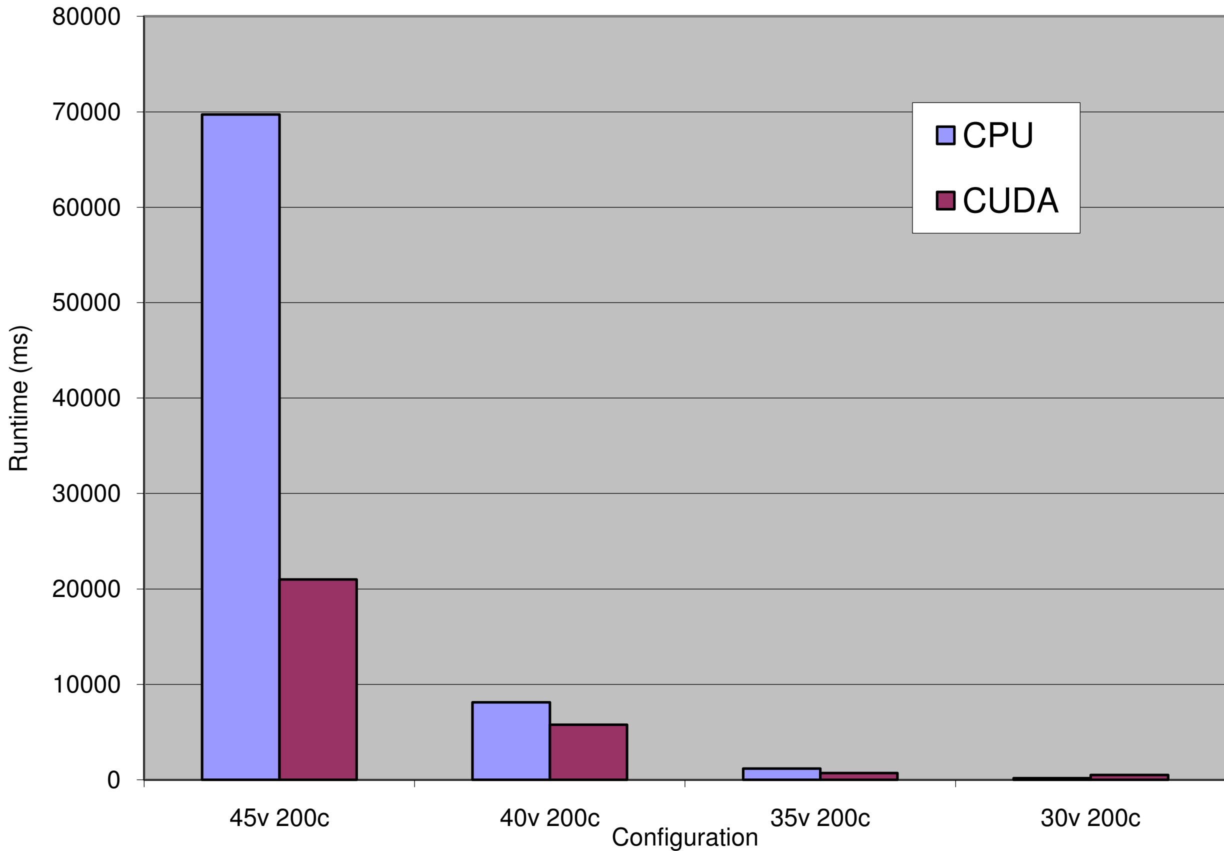

Figure 4. Baseline CPU vs. CPU/GPU hybrid solution. *v* stands for number of variables, *c* stands for number of clauses.

Work-stealing not only causes each thread to be more efficient, but also enables more parallelism. Since increasing the number of jobs without work-stealing also decreases the efficiency of each job, it is not possible to have a large number of jobs since many processing units will be sitting idle after the jobs terminate early. Work-stealing solves this problem and thus allows the number of jobs to scale much better. Thus, with work-stealing, we can run a much greater number of jobs, which allows us to run more threads in CUDA, which allows us to hide latency much more effectively.

D. Final Algorithm vs. CPU

Since we did not have time to implement many of the advanced optimizations for DPLL, our algorithm scaled very poorly. Thus, we were only able to test on fairly small datasets. A real SAT problem would be orders of magnitude larger than our sample testbenches. As demonstrated in Figure 4, even at such small input sizes, we see good speedup in the 45 clause case. Recall that SAT problems scale as a function of the clause squared. Obviously, our algorithm will see much better speedups with larger input sizes.

V. Work Dynamics

For this project, Cas wrote the SAT software implementation and parts of the CUDA algorithm. Dan wrote the random DIMACS CNF generator, the parallel DPLL algorithm, the work stealing algorithm, and the paper. Bhargavi wrote the CUDA kernels and extended them to support various optimizations. Everyone assisted on the debug of the code as well as the data collection.

VI. Conclusion

In this paper, we presented a novel approach to dividing the Boolean satisfiability problem into two components: control, which is performed by the CPU, and data, which is performed by the GPU. Our results demonstrate the effectiveness of our solution. We implemented several optimizations to decrease the effect of the GPU to CPU communication over the slow PCI-E bus, as well as load balancing algorithms to efficiently scale to a large number of independent tasks. These changes improved our performance dramatically. The separation of DPLL into several

independent tasks gained us about 10x in performance, while implementing our work stealing algorithm improved performance 3x further. Implementing our optimizations on top of our infrastructure was fairly straight-forward, and no doubt implementing the standard SAT optimizations should be similarly trivial.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

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

[3] Silva, J. P. and Sakallah, K. A. 1996. GRASP—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international Conference on Computer-Aided Design* (San Jose, California, United States, November 10 - 14, 1996). International Conference on Computer Aided Design. IEEE Computer Society, Washington, DC, 220-227.

[4] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference* (Las Vegas, Nevada, United States). DAC '01. ACM, New York, NY, 530-535.

[5] Huang, J. 2007. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international Joint Conference on Artifical intelligence* (Hyderabad, India, January 06 - 12, 2007). R. Sangal, H. Mehta, and R. K. Bagga, Eds. Ijcai Conference On Artificial Intelligence. Morgan Kaufmann Publishers, San Francisco, CA, 2318-2323.

[6] Bohm, M., and Speckenmeyer, E. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 1996.

[7] Feldman, Y., Dershowitz, N., and Hanna, Z. Parallel Multithreaded Satisfiability Solver: Design and Implementation, Electronic Notes in Theoretical Computer Science, Volume 128, Issue 3, *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification* (PDMC 2004), 19 April 2005, Pages 75-90, ISSN 1571-0661.

[8] Compute Unified Device Architecture (CUDA) Programming Guide. http://developer.nvidia.com/object/cuda.html: NVIDIA, 2007.

[9] Plaisted, D., and Greenbaum, S. A structure-preserving clause form translation, *Journal of Symbolic Computation 2* (1986), 293-304.