9-2010

Application of CUDA in the Boolean Domain for the Unate Covering Problem

Eric Paul
Portland State University

Bernd Steinbach
Freiberg University of Mining and Technology

Marek Perkowski
Portland State University

Citation Details

Paul, Eric, Bernd Steinbach, and Marek Perkowski. "Application of CUDA in the Boolean Domain for the Unate Covering Problem." Boolean Problems, Proceedings of the 9th International Workshops on Boolean Problems. 2010.

Application of CUDA in the Boolean Domain for the Unate Covering Problem

Eric Paul[1], Bernd Steinbach[2], and Marek Perkowski[3]

[1,3]Portland State University
Department of Electrical Engineering
Portland, Oregon, USA
email: epaul@cecs.pdx.edu[1], mperkows@cecs.pdx.edu[3]

[2]Freiberg University of Mining and Technology
Institute of Computer Science
D-09596 Freiberg, Germany
email: steinb@informatik.tu-freiberg.de

Abstract

NVIDIA's Compute Unified Device Architecture (CUDA) is a relatively-recent development that allows to realize very fast algorithms for several Constraint Satisfaction and Computer Aided Design tasks. In this paper we present an approach to use Graphics Processing Units (GPU) and CUDA for solving Unate Covering Problem, a practical problem related to SAT. In particular we present a CUDA-enabled Petrick Function Minimizer. We compare the performance of a pipeline-processor (CPU) and a parallel processor (GPU) implementation of the matrix-multiplication method for solving unate covering problems.

1 Introduction

It is well-known that many important problems can be reduced to SAT [1], [3], [4], [5], [6], [7], [8], [9], and [16]. An Oracle is a Boolean function that evaluates some set of solution candidates with yes or no output. In graph coloring [13], an oracle is commonly used to determine if there exists a valid coloring below some threshold—the decision. Finding the minimum coloring involves running the solver iteratively with lower thresholds until a "no" is produced. It is often faster to solve the decision problem than it is to find the minimal coloring for this type of problem.

The SAT oracle [13] is also a Boolean function, just a Product of Sums (POS) of literals. The feature of interest for the SAT solver is an assignment of variables such that the objective function is satisfied. This is inherently more complex than the simple decision problem.

A special type of SAT problems is given when each literal appears in a fixed polarity in the POS. This restriction to a unate Boolean function simplifies the SAT problem, because a trivial solution, where all literals are assigned with the value of the given polarity, can be given immediately. In this paper we solve the more difficult Unate Covering Problem (UCP) where only solutions are searched having minimal cost (number of literals required) that satisfy the given Boolean equation. The UCP is encountered in practice in many places, such as in SOP (Sum of Product) minimization of Boolean functions, minimization of finite state machines, test minimization and scheduling/allocation, logistics, to name just a few.

It is main aim of this paper to explore the application of CUDA in the Boolean domain. We have chosen the UCP as example of the task to solve because solutions to this task are required in logic synthesis, the known solution algorithms have an exponential complexity, and the number of variables is typically much larger than the number of variables in the basic functions explored in this paer. Due to our aim of demonstrating CUDA application design, we selected a simple algorithm and compared the runtime between the CPU-only and GPU-acccelerated implementations. We are certain that more efficient approaches for the UCP could

be developed which will be studied in the future.

Our results indicate that an algorithm solving arbitrary UCPs with matrix-multiplication on CUDA results in nearly-linear speedup . One factor predicting the amount of speedup for any given problem is the problem size itself. In general, the larger the problem, the greater the speedup. For the experiments in this paper, we found that speedup was achieved on problems with as few as 16 literals (fig. 5(c)).

Our paper is structured as follows. In Section 2, we discuss the UCP problem and how a matrix-multiplication algorithm can be used as a solver. For comparison, we summarize the CPU-only approach in Section 3. We explain the CPU + GPU implementation using CUDA in more detail in Section 4. All programs are written in ANSI C. Our experiments are restricted to a range that allows to observe the expected speedup. The experimental results are given in Section 5. In Section 6, we conclude the presented approaches and explain some directions of our future work.

2 The Unate Covering Problem

2.1 Unate Functions

The Unate Covering Problem (UCP) is an important simplification of general SAT, in that it simplifies solving techniques. To see this, we first set forth a working definition for unate functions using the simple derivative of the Boolean differential calculus, see [18] and section four in [11].

Definition 1 ***Unate Functions***
For some arbitrary Boolean function of n variables, $f(x_1, x_2, \ldots, x_i, \ldots, x_n)$, f is considered positively unate in x_i iff

$$\overline{x}_i \wedge \frac{\partial f(x_1, x_2, \ldots, x_i, \ldots, x_n)}{\partial x_i} = 0 \ . \tag{1}$$

Similarly, the function f is negatively unate in x_i iff

$$x_i \wedge \frac{\partial f(x_1, x_2, \ldots, x_i, \ldots, x_n)}{\partial x_i} = 0 \tag{2}$$

holds. If f is positively (negatively) unate in all x_i, then we say that the function is positively (negatively) unate. If either (1) *or* (2) *holds for each variable x_i, then the function f is called unate. If this condition fails (at least one variable is binate) for any variable, then we say that the function is not unate; it is binate.*

If the function we wish to solve is either negatively- or positively-unate, then the solution we seek is not concerned with the polarity of the variables in the solution, we are only interested in which literals appear in a given solution. Thus, the technique presented here works for unate problems of either polarity. This further reduces the problem to subset selection, which is computationally simpler than variable assignment, as in general SAT.

2.2 UCP Formulation

Let us take the SOP minimization of Boolean function as an example of a UCP. The associated unate covering problem can be expressed as a prime implicant (PI) chart. In Table 1, we see such a chart.

Expressing Table 1 in the form of a Product-of-Sums (POS) Boolean function, we find the following function:

$$P = (L \vee M) \cdot (N \vee Q) \cdot (L \vee N) \cdot (K \vee R) \cdot K \cdot (M \vee R) \cdot (K \vee Q) \cdot (K \vee S) \cdot S \ . \tag{3}$$

The Boolean-expression form of (3) is referred to as a *Petrick* function. There are many ways to find an assignment of a product of the prime implicants such that the function evaluates to true (all clauses are true). One such method it tree branching. Paths to the leaves of the tree resulting from branching on the expression represent a product of literals that satisfies the chart.

Table 1: The Prime Implicant Chart for Equation (3).

Prime Implicants	Clauses								
	1	2	3	4	5	6	7	8	9
K				X	X		X	X	
L	X		X						
M	X					X			
N		X	X						
Q		X					X		
R				X		X			
S								X	X

Prime Implicants	Clauses	
	1	2
A	X	
B	X	X
C		X

(a)

$$N = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$$

(b)

Figure 1: Prime implicant chart (a) and matrix (b) representations of the Petrick function in Equation (5).

The minimal solutions are the shortest path to the leaves; we have a single minimal solution of cost 4,

$$P = K \cdot S \cdot M \cdot N \ . \tag{4}$$

Well-known methods for implementing a tree-search algorithm requires recursion and sequential data structures: a stack for depth-first search, a queue for breadth-first search, or both for a hybrid method like iterative deepening. Our target hardware platform in this case is an NVIDIA Compute-Unified Device Architecture-enabled graphics processing unit (CUDA GPU). At the top level, a single CPU thread launches a multi-threaded GPU routine referred to as a *GPU kernel*. Recursions are not allowed in GPU kernels [2].

This intuitively makes sense due to the fact that we wish to operate on blocks of data in parallel, which goes against the sequential nature of recursion. More importantly, the data structures themselves in tree search—and recursion in general—are not "data-parallel," i.e. stacks and queues only allow access to the ends of the structure.

In order to achieve an appreciable amount of speed up with CUDA, the underlying data structures should be random-access in nature. One-, two-, and three-dimensional arrays are one of the primary data structures used in CUDA programming. One particular method of computation that is well suited for CUDA execution is matrix multiplication. Seeking a quick implementation, we chose to implement the solver with Boolean matrix multiplication. Obviously, there are other ways to implement the solver, but this is one simple way that produced a quick prototype and allows us to compare the runtime between a CPU- and an equivalent GPU-implementation.

2.3 Solving UCP with Boolean Matrix Multiplication

Let us consider the algorithm with a simple example. In eq. (5) we have a 3-literal, 2-clause Petrick function. The associated prime implicant chart and its matrix representation are shown in Figure 1.

$$P = (A \vee B) \cdot (B \vee C) \tag{5}$$

We can see by inspection that the minimal-cost solution is B, since chosing that literal satifies both clauses. In the matrix representation of eq. (5), a 1 in a clause column indicates that the associated literal appears in that clause. Whether or not this literal appears positively or negatively is irrelevant since this is a unate function. Table 2 shows the truth table for this function.

Table 2: Truth table for the Petrick function in Equation (5).

Input	Clauses		Output	Literal
ABC	$A \vee B$	$B \vee C$	P	Cost
0 0 0	0	0	0	0
0 0 1	0	1	0	1
0 1 0	1	1	1	1
0 1 1	1	1	1	2
1 0 0	0	0	0	1
1 0 1	1	1	1	2
1 1 0	1	1	1	2
1 1 1	1	1	1	3

Again, the *input* patterns denote whether or not the associated literal appears in a solution candidate. From this we can explicitly see five solutions: B is the minimal solution of cost = 1. BC, AC, and AB are solutions of cost 2, and ABC is the maximum-cost solution with 3 literals. We sort the truth table according to literal cost of the input patterns, then define matrix M that represents the truth table (all possible solutions are represented). Finally, M is Boolean-multiplied by N from fig. 1 (b). The resulting output matrix P is produced, as indicated in eq. (5):

$$M \times N = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = P \tag{6}$$

The Boolean matrix multiplication algorithm is the same as arithmetic matrix multiplication except that we replace the scalar multiplication operator with the Boolean AND operator, and replace the arithmetic sum operator with the Boolean OR operator. Any given element in the output matrix, $P(i, j)$, represents whether or not the input pattern (literals included in the candidate solution) from row i in M satisfies the clause in column j in N.

Take row 2 of the output matrix as an example: multiplying the second row of M with the first column of N, we obtain zero for the Boolean OR of the three Boolan ANDs, so a zero is entered in the associated output position, $P(2, 1)$. To complete the row in the output matrix, we perform the same operations with the second clause in the function (second column of N) with the same input pattern. This time the output element is a logic 1 since there was a match—a 1 in corresponding positions—between the input pattern and the second clause. Thus a 1 is stored in $P(2, 2)$.

The minimal condition for a single solution candidate satisfying the constraint matrix is just a row in the output matrix containing no zeros. The first row in the output matrix with no zeros is the third, which is associated with the input pattern 010, representing the solution B. We also have non-zero-containing rows below row 2. Since the rows in the all-possible-solutions matrix M are sorted by cost, the rows of the output matrix are ordered by the cost of their associated solution candidate. The non-zero rows in P are associated with the input patterns $\{010, 011, 101, 110, 111\}$. These input patterns have a cost of $\{1, 2, 2, 2, 3\}$; sorting the input space by literal cost is key to finding minimal solutions first.

One could argue that that if a zero is produced in the output matrix, the search for that solution cadidate should be cut off. While this would be efficient for a sequential solution, interupting CUDA matrix multiplication on a per-row basis is not only hard to design, but would be an ineffective CUDA-programming tactic. When designing for CUDA, we want to parse our algorithms into bulk operations that can be performed without branching, i.e. *if-then* constructs.

The algorithm we implement in the software performs this multiplication without regard to whether or not a zero is produced in the output matrix. We just perform the multiplication, and then evaluate the output matrix in a separate routine. This allows us to use complete matrix

multiplication as our solver engine. Further, it allows us to test arbitrary UCPs that can be represented as different instantiations of matrix N, regardless of the features of the UCP constraints themselves. In other words, different UCPs of the same size will have a predictable—ideally identical—evaluation time, no matter how difficult the problem is to solve. When comparing problems of the same size, those with fewer good solutions are considerd more difficult; our technique is indifferent to this metric.

3 CPU-Only Appraoch

Since these problems can get quite large, we decided to implement the reading of the PI chart and the reporting of solutions via file I/O using ANSI C. The basic algorithm our program implements is indicated in Algorithm 1.

Algorithm 1 Top-Level Sequential (CPU-only) UCP Algorithm

1: read the PI chart from input file (initialize N)
2: create ordered solution space based on number of literals in PI chart (initialize M)
3: perform sequential Boolean matrix multiplication ($P = M \times N$)
4: search output rows of P for satisfying rows
5: report minimal solutions to output file

The primary data structure in use for the software is a one-dimensional array of Boolean flags. This is very efficient in that it only uses a single bit per matrix element. Matrices are stored in row-major order, meaning matrix rows are stored in the array sequentially.

Both simple examples were run for the example Petrick functions in eq. (5) and eq. (3); minimal solutions of cost 1 and 4 were found, respectively. This verifies the correctness of the algorithm, and verifies the result obtained by branching, indicated in [13], and [10]. Results of larger benchmarks are given in Section 5.

4 GPU-Enabled Appraoch

The CUDA-enabled algorithm is nearly identical to the sequential version. In Algorithm 2, we added memory transfers between the GPU and main system memory. The memory transfers are not a zero-time phenomenon; they are in many cases one of the main bottlenecks in any CUDA program. There are ways to combat the memory transfer bottleneck (page-locked and/or pinned host memory [2], partitioning, etc.), but that is a topic for future research.

Algorithm 2 Top-Level Parallel UCP Algorithm

1: read the PI chart from input file (initialize N)
2: create ordered solution space based on number of literals in PI chart (initialize M)
3: transfer input matrices to the GPU
4: perform CUDA Boolean matrix multiplication ($P = M \times N$)
5: transfer results from GPU to main system memory
6: search output rows of P for satisfying rows
7: report minimal solutions to output file

4.1 CUDA Matrix Multiplication

The matrix product $A \times B = C$ is represented graphically in Figure 2. We parse the input and output matrices into sub-matrices represented by the yellow blocks. Note that this block size is square, and is consistent for both of the input matrices and the output matrix. The input matrices A and B have dimensions $wA \times hA$ and $wB \times hB$, respectively, where $hB = wA$.

The chosen block size must cover—exactly—the dimensions of the matrices we wish to multiply. Meaning that the height and width of both matrices must be an integer multiple of the chosen block size. The key feature of this blockwise algorithm is that each GPU thread computes a single output-matrix element within the respective block.

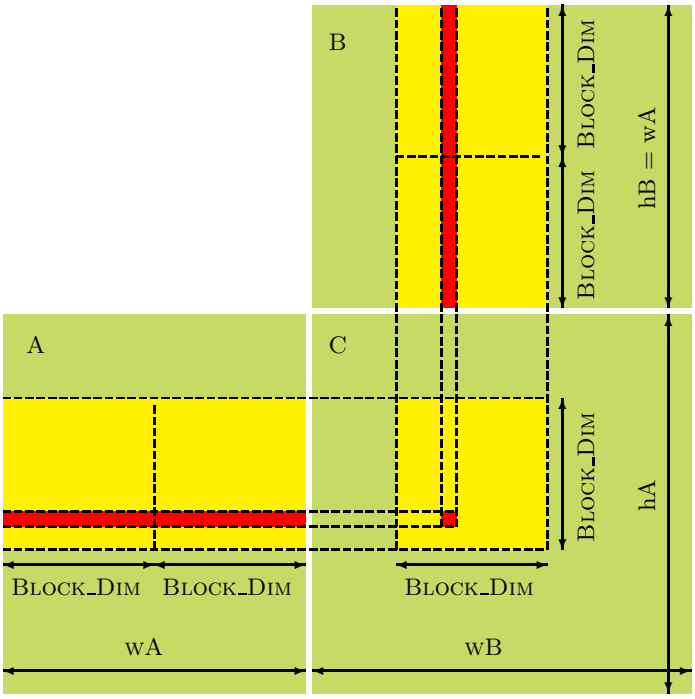


Figure 2: Matrix multiplication on CUDA

Before the kernel execution begins, we copy the input matrices entirely to the GPU's global memory in a single asynchronous memory transaction for each input matrix. Once there, it is available to all blocks in the kernel. Then, the output matrix is partitioned into sub-matrices C_{sub} (yellow block, fig. 2). These sub-blocks of the output matrix are the GPU kernel thread blocks that are calculated in parallel. More precisely, all the elements in any given output submatrix are concurrently computed on the GPU within a single thread block, and as many blocks are computed in parallel as the specific CUDA device will allow (see sec. 4.2.1).

This is effectively accomplished utilizing shared memory. For a single output sub-matrix (yellow sub-block of C in fig. 2), the kernel block first loads the left-most sub-matrix of A, and the uppermost sub-matrix of B in fig. 2 into the block's local shared memory. Then, the Boolean multiply-and-accumulate is performed with those input sub-matrices. We iteratively continue this until all the required multiply-and-accumulate operations have been performed to compute a single sub-block.

Since threads within a block, along with blocks themselves are calculated in parallel, we end up with multiple copies of distinct ranges of global memory residing in separate shared memory blocks. While this may seem redundant and inefficient, it speeds up the process since we can then compute all (or as many as the device will allow) the sub-matrices of the output in parallel; we do not induce bank conflicts since each thread block has it's own copy of the data required for the computation.

As presented earlier, the matrices we wish to multiply on the device must be sized such that the number of rows and columns in the input matrices must be an integer multiple of the block size. This creates an unwanted connection between input size and the block size we choose for the GPU kernel. Again, we're in pursuit of a quick prototype for this basic research, so we move forward with the restricted input sizes. To better understand CUDA in general, and how it applies to the UCP solver, we now explore GPU kernel design basics.

4.2 CUDA-based UCP Solver Implementation

4.2.1 GPU Kernel Design

Part of utilizing CUDA effectively involves designing kernel configurations that strike a balance between dense thread blocks and masking memory latency [2]. Most CUDA devices can handle

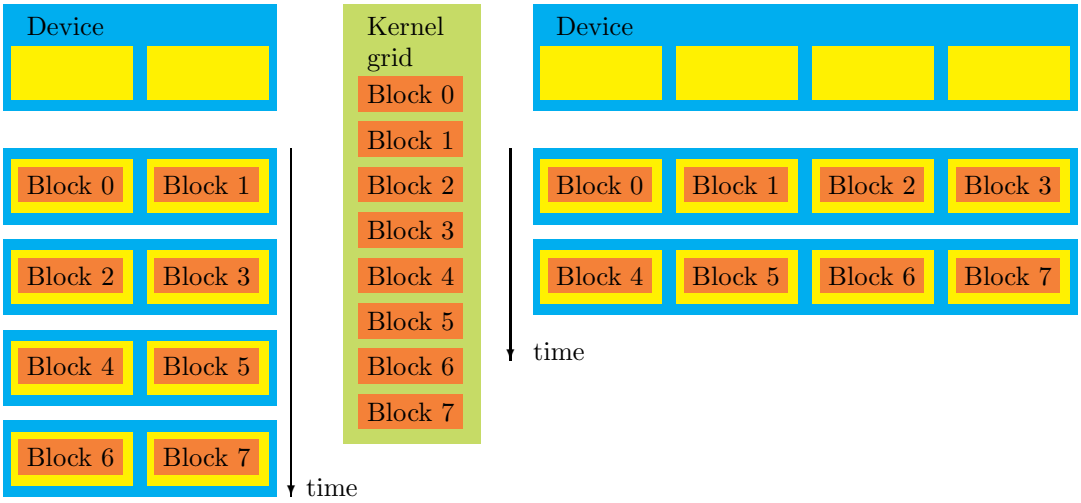


Figure 3: Transparent GPU Kernel Grid Scaling

a maximum of 512 threads per block. Pushing the thread density up against this limit reveals memory latency (bank conflicts), while lowering the thread density to near 1 doesn't efficiently utilize computing resources on the device (under-occupation). In either case, we suffer significant performance degradation. Further, we want the thread count per block to be a multiple of the warp size [2]—32 for most CUDA devices. NVIDIA documentation suggests that thread density in general should be set somewhere between 128 and 256 threads per block to achive the balance of device occupancy and latency masking.

Taking this into account, we chose a block size of $8 \times 8 = 64$ threads per block (BLOCK_DIM = 8 in fig. 2). This is an integer multiple of the 32-thread warp size for our devices, and is much less than the 512-thread limit for kernel blocks on our device. Although we know that 64 threads per block does not adequately occupy the device, choosing this size allowed us to solve UCPs whose literal and clause counts are a multiple of 8. Again, we're looking for a quick prototype, future work would certainly consider flexible block sizing—perhaps dynamic, based on input size. Ultimately, we'd like to decouple input size from GPU kernel configuration.

Setting the block sizes as such, we then end up with some number of blocks of threads to execute in the GPU kernel grid. Depending on input size and the specific device we are working with, we may or may not be able to process all blocks in concurrently. One of the key design goals of the CUDA architecture and programming model is to facilitate transparent scaling of a GPU kernel grid across many devices—without recompiling.

Given the number of blocks that are required to cover the data, and the thread density of those blocks, the CUDA runtime subsystem will manage and execute as many blocks concurrently as possible [2] (see Figure 3). Clearly, larger devices can execute more blocks in parallel than smaller devices.

5 Experimental Results

We built a random input file generator to create a series of PI charts. For comparison purposes to the GPU implementation (see Section 4), we generated input files with 8 and 16 literals, varying the clause count between 8 and 256. We then ran the same source program compiled for the two machines indicated in Table 3. The timing results are indicated in Table 4.

For the 8-literal cases, we see an approximately-constant execution time for both machines. More than 95% of the execution time for those small cases is spent on generating the ordered solution space. It is not until the 16×32 case that the algorithm spends more time on matrix multiplication than generating the solution space. This is somewhat expected in that the 8-literal problems only have 256 possible solutions, corresponding to matrix rows in the input space. This is a very small matrix in terms of memory requirements. Hence, the supporting tasks strongly dominate the main task of matrix multiplication. Further, the cost of memory transfer outweighs the performance gain from using the GPU, so GPU utilization for small

Table 3: Test machines

	Machine A	**Machine** B
CPU	Intel Core i7 940 (2.93 GHz)	Intel Core 2 Duo (2.66 GHz)
RAM	12 GB	4 GB
OS	Windows 7 (64 bit)	Mac OS 10.6.4 (64 bit)
GPU	NVIDIA GeForce 9600 GT	NVIDIA GeForce 9600M GT
global memory	512 MB	256 MB
cores/SM count	64/8	16/2
compute capability	1.1	1.1

Table 4: Timing results in milliseconds

		CPU-only		CPU-GPU	
Literals	Clauses	Machine A	Machine B	Machine A	Machine B
8	8	17.503914	7.493989	54.658726	43.221001
	16	17.733423	7.595000	54.835838	42.880010
	32	17.965380	8.541000	54.090714	41.702999
	64	18.462826	8.506000	55.762260	42.201000
	128	18.562735	8.989000	54.305553	43.050999
	256	19.586273	10.229000	54.915833	42.980000
16	8	46.765251	60.618999	62.663292	61.612000
	16	70.407600	77.098999	70.512749	77.869003
	32	131.675659	119.874001	78.657043	108.890999
	64	222.917694	247.671005	93.964371	172.565002
	128	421.712708	433.269012	138.112427	305.154999
	256	810.593933	910.721985	236.531815	431.035094

problems slow down the total run time, which can clearly be seen in fig. 5(a). However, the maximal run time of less than 60 milliseconds is so short that no special fitted algorithms are needed.

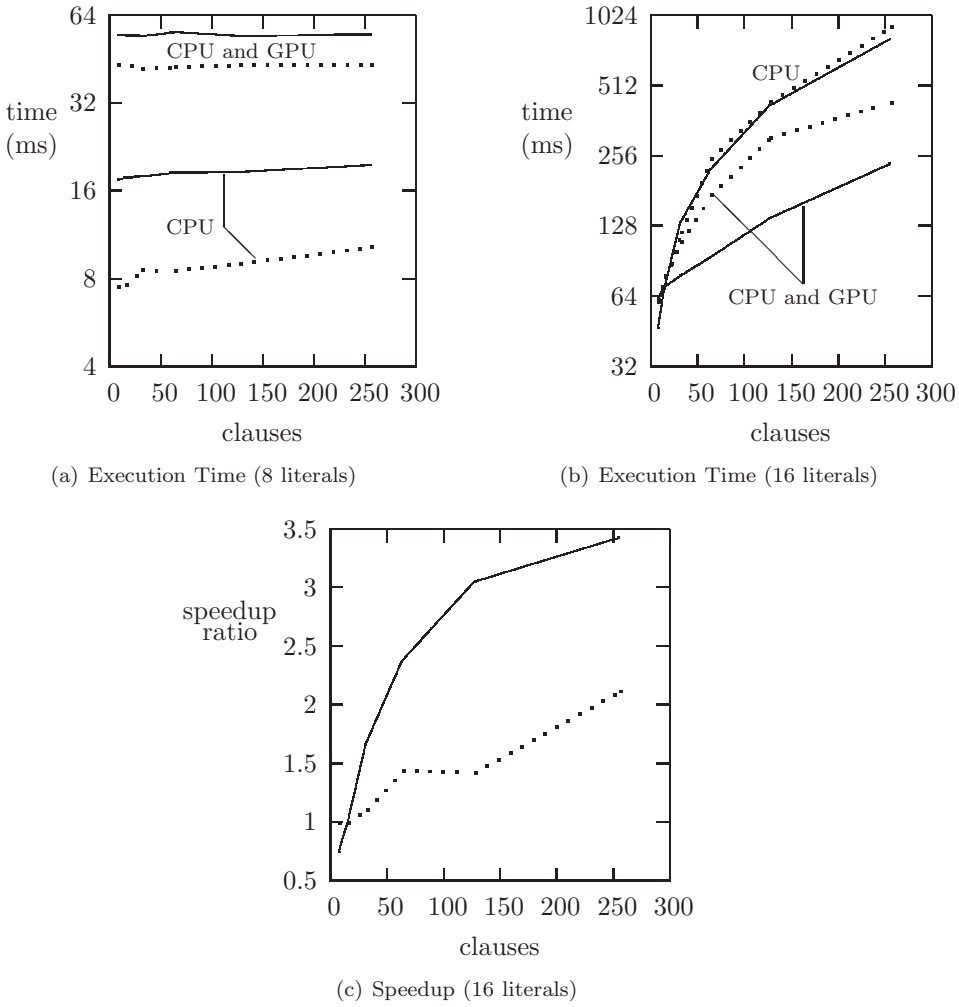
The 16-literal cases exhibit the expected behavior of execution time growing with input size, and the execution time is nearly identical between the two machines for the CPU-only case. This is somewhat expected in that the CPU-only program is a single-threaded application running on a single core of CPUs with nearly identical features per core (see Table 3). The GPU-enabled versions exhibit a slower increase of computation time with respect to problem size, as seen in the plot of fig. 5(b).

We can more precisely examine the computational benefit yeilded by the CUDA implementation for the 16-literal case with the speedup-ratio plot presented in fig. 5(c). Speedup is defined as the ratio of the CPU-only execution time to the GPU-enabled execution time. We clearly see the effects of transparent scaling in this plot, as the 9600 GT's speedup ratio grows much faster than the 9600M GT's—a direct result of the larger device being able to compute more sub-blocks of the output matrix in parallel.

It should be noted that the CUDA devices these tests were performed on were supporting display (rendering screen output) while computing. Thus, the speedup metrics are likely lower than what a computation-only (dedicated) CUDA device could deliver. In any case, our results indicate that the largest problem (16×256), the runtime is reduced by a factor of 3.4. This reinforces our earlier idea that the larger the problem, the more computational benefit we yeild by utilizing the CUDA architechture.

6 Conclusions and Future Work

We presented initial work on a basic UCP-solver parallelization. This simple method could likely be extended to work with more general constraint-satisfaction and oracle-like problems. The idea is that if we understand how to effectively use the hardware, we can map that knowledge into solving more complex problems. Although the problems solved here are very small with



(a) Execution Time (8 literals)

(b) Execution Time (16 literals)

(c) Speedup (16 literals)

Figure 4: Timing results: machine A (solid lines) / machine B (dotted lines)

respect to standard benchmarks, this early work clearly indicates computational benefits can easily be obtained with CUDA.

Matrix multiplication is an excellent example problem to explore the CUDA architecture and programming model. Although transparent scaling allows us to abstract away the capacaties of specific CUDA devices, we must learn to think in parallel with the CUDA architecture in mind: both the problem we wish to solve and the general CUDA architecture must be taken into account to develop effective CUDA-enabled applications.

Future work will include learing more about the CUDA memory model. We did not explore utilizing thread-local registers as a memory optimization as outlined in [19]. Flexibility improvements could be made by decoupling the GPU kernel configurations from input sizes. Along those same lines, a partitioning scheme could allow us to problems larger than 16 literals by searching lower-cost solution candidate groups sequentially.

Finally, we did not implement any heuristics when developing the solution candidate space. There are many well-known methods that have been published [10], [11], [12], [13], [14], [15], and [17]. Since this was a CUDA exploration, rather than a SAT project, we largely ignored the available literature regarding heuristic methods. The sheer size of the memory requirements utilizing this brute-force method clearly indicate that implementing heuristics would be an effective tactic in creating a more efficient solver.

References

[1] Cook, S. *The Complexity of Theorem-Proving Procedures.* Proceedings Third Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–198.

[2] *Current CUDA Programming Guide* see: http://www.nvidia.com/object/cuda_get.html

[3] Davis, M. and Putnam H. *A Computing Procedure for Quantification Theory.* Journal of the ACM, Volume 7, Issue 3, New York, 1960, pp. 201–215.

[4] Davis M., Logemann, G. and Loveland M.D. *A Machine Program for Theorem Proving.* Communications of the ACM 5, New York, 1962, pp. 394–397.

[5] Gu J., Purdom P., Franco J. and Wah B. *Algorithms for the Satisfiability (SAT) Problem: A Survey.* Preliminary version, 1996, see: http://citeseer.nj.nec.com/56722.html.

[6] Johnson M. and Posthoff Ch. *TRISAT - A SAT - Solver Using Ternary-Valued Logics.* 14th International Workshop on Post-Binary ULSI Systems, Calgary, Canada, 2005, pp. 1–17.

[7] Johnson G. *Computers and Intractability: A guide to the theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979, ISBN 0-7167-1045-5.

[8] Karp R. M. *Complexity of Computer Computations.* In: Miller R. E. and Thatcher J. W. (editors): Reducibility Among Combinatorial Problems, New York, Plenum Press. 1972, pages 85–103.

[9] Levin, L. *Universal'nye Perebornye Zadachi.* Problemy Peredachi Informatsii Volume 9 Issue 3, 1973, pp. 265-266. English translation: Universal Search Problems. In: Trakhtenbrot B. A. A Survey of Russian Approaches to Perebor (Brute-Force Search) Algorithms. Annals of the History of Computing Volume6 Issue4, 1984, pp. 384–400.

[10] Paul, E. *M.Sc. thesis in preparation.* PSU, 2010.

[11] Posthoff, Ch. and Steinbach, B. *Logic Functions and Equations - Binary Models for Computer Science.* Springer, Dordrecht, The Netherlands, 2004.

[12] Perkowski M. and Mishchenko, A. *Logic Synthesis for Regular Layout using Satisfiability.* in: Proceedings of the 5th International Workshop on Boolean Problems, 2002, pp. 225–232.

[13] Perkowski, M. et. al. *Using Veloce for Oracle Implementation.* PSU. Submitted 2010.

[14] Posthoff, Ch. and Steinbach, B. *A Multi-Processor Approach to SAT-Problems.* 7th International Workshop on Boolean Problems, 19th - 20th of September 2006, Freiberg University of Mining and Technology, Freiberg, Germany, 2006, pp. 49–62.

[15] Posthoff, Ch. and Steinbach, B.: *SAT-Problems - New Findings.* in: Proceeding of the 6th WSEAS International Conference on Data Networks, Communications, Computers, Trinidad and Tobago, November 5–7, 2007, pp. 339–344.

[16] Rish I. and Dechter R. *Resolution versus Search: Two strategies for SAT.* SAT2000: Highlights of Satisfiability Research in the Year 2000, ; Vol. 63 of Frontiers in: Artificial Intelligence and Applications, Gent, I., et al, (Editors) IOS Press, ISBN 1-58603-061-2, 2000, pp. 215–259.

[17] Schöning, U. *A probabilistic algorithm for k-SAT and constraint satisfaction problems.* Proceedings of the 40th Symposium on Foundations of Computer Science, IEEE, October 1999, pp. 410-414.

[18] Steinbach, B. and Posthoff, Ch.: *Boolean Differential Calculus.* in: Sasao, T. and Butler, J.T.; Progress in Applications of Boolean Functions Synthesis Lecturers on Digital Circuits and Systems # 26. Morgan & Claypool Publishers, San Rafael, CA USA, 2010, ISBN 978-1-60845-181-4, pp. 55-78, and 121-126.

[19] Kirk, David B. and Hwu, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers, 2010, 0-1238-1472-3.