Michael Wang (mswang)
Ran (Charles) Guo (rang)
May 11, 2012

15-418 Final Project Writeup
Parallel k-SAT Solver

I. Summary

We've implemented a parallel k-SAT Solver using CUDA on the NVIDIA GPUs in the Gates 5th floor lab. Our solution proved to be faster than the serial implementation we've implemented on the GPUs, and our parallel solution also scaled well as the number of threads increased.

II. Background

The following is a review of some terms and definitions.

Consider a set of boolean variables $x_1,x_2,...,x_n$

A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
A **clause** is a disjunction of literals. E.g. ($x_1 \vee x_2 \vee \neg x_4$) is a clause
A formula in **conjunctive normal form (CNF)** is a propositional formula which is a conjunction of clauses. E.g. ($x_1 \vee x_2 \vee \neg x_4$) ^ ($x_3 \vee x_4$) ^ ($\neg x_5$)

The SAT problem we're solving is

"Given a CNF formula, is there a truth assignment to the variables such that the formula evaluates to true?"

Some examples to help illustrate...
Example 1:
($x_1 \vee x_2 \vee \neg x_4$) ^ ($x_2 \vee \neg x_3$) ^ (x_5)
is satisfiable if you assign $x_1,...,x_5$ to be true

Example 2:
($x_1 \vee \neg x_2$) ^ ($\neg x_1 \vee x_2$) ^ ($\neg x_1 \vee \neg x_2$) ^ ($x_1 \vee x_2$)
is not satisfiable because there is no assignment of x_1 and x_2 that makes the formula true

Why should you care about this problem?
It is the first known example of an NP-Complete Problem. These means that other NP problems can be reduced to them. It arises in many applications involving hardware and software verification and correctness.

III. Serial Approach

We first wrote a serial version of the k-SAT solver on the CPU. The simplest way to solve this was to try all possible assignments of boolean variables $x_1,x_2,...,x_n$. Namely, the algorithm is

1. Choose a unassigned literal
2. Assign a truth value to it
3. Simplify the formula based on this assignment
4. Recursively check if the simplified formula is satisfiable
    - If yes, then the original formula is satisfiable
    - If no, then we do the same recursive check using the opposite truth value

The simplification step (3) essentially removes all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses.

For example, if the formula is
$(x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_3) \wedge (x_5)$
and the we choose to assign x_1 to be true, then the simplified formula would be
$(x_3) \wedge (x_5)$

Basically the first clause was removed because the clause contained one literal that evaluated to be true, and the $\neg x_1$ in the second clause evaluated to be false, and thus it was removed inside the clause.

We can see that the running time of this naive algorithm is $O(2^n)$ where n is the number of boolean variables. This is because there are n variables and each of them can be assigned true or false, and we are testing all possible assignments.

After implementing this naive solution, we've decided to improve upon this code by implementing the **Davis-Putnam-Logemann-Loveland (DPLL)** algorithm. It was first introduced in 1962, but it still is a highly efficient procedure for solving the SAT problem and forms the basis for most efficient complete SAT solvers.

The DPLL algorithm improves upon the naive algorithm described earlier by using two rules at each recursive step: **Unit propagation** and **Pure literal elimination**.

Unit propagation: If a clause in the formula is a unit clause (meaning it contains only one unassigned literal, then the clause can only be satisfied by assigning the necessary value to make this literal true. For example, if the formula is
$(x_1 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_4)$
then the unit propagation rule would assign x_4 to be false and the resulting formula would be
$(\neg x_1 \vee x_3)$

Note that this rule eliminates a large part of the search space, specifically cuts at least half of the search space for each unique literal in a unit clause.

Pure literal elimination: If a variable occurs with only one polarity in the formula, it is called pure. Pure literals can be assigned so that all clauses containing them are evaluated to be true.
For example, if the formula is
$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)$
then the pure literal elimination rule would assign x_3 to be true and the resulting formula would be
$(x_1 \vee \neg x_2 \vee x_4)$

In the DPLL algorithm, unsatisfiability of a partial assignment would be detected if one clause became empty, or in other words if the variables have been assigned in a way that makes all the literals in one clause to be false. Satisfiability would be detected when all the variables have been assigned without generating the empty clause, or in other words if all clauses have been satisfied given the assignment. Unsatisfiability of the entire formula would be detected only after an exhaustive search.

Note that the worst-case running time of the DPLL algorithm is still exponential, but these two rules greatly reduce the search space in practice.

IV. Parallel Approach

Our parallel approach was to parallelize the serial version of the DPLL algorithm.

For our initial implementation, we attempted to solve the problem using the recursive algorithm described in the previous section. CUDA supports recursion on the Gates' GPUs. However, recursion using CUDA is not as efficient compared to recursion on a CPU due to limited amount of memory available for recursion on the GPU. Because the DPLL algorithm needs to store a different working set at each recursive level, the GPU would run out of memory with which to perform recursion. Furthermore, recursion is generally more expensive on the GPU compared to the CPU due to factors such as a larger number of registers. A large number of registers means that the amount of memory needed to be saved and restored to the stack during recursion will be larger.

Because of these problems associated with recursion in CUDA, we decided to simulate a stack in a non recursive function in CUDA. In our solution, we save a partial assignment of values to variables to a stack in local memory. In order to recover the working set that we would have saved on our stack had we used recursion, we recompute our working set using the saved partial assignment. We chose this approach because the amount of memory needed to save the partial assignment to a stack in local memory is small. If we had chosen to save our working set in global memory, our memory footprint would have been much larger. Furthermore, saving our working set in global memory could have potentially created an implementation that was bandwidth bound because each individual thread would continually be saving and writing to global memory. Although only saving the partial assignment means that we need to recompute the working set, we avoid becoming bandwidth bound due to global memory. An added benefit of managing recursion this way is that the size of the problem we can store in local memory increases because the amount of space needed to store the partial assignment is small compared to the size of the entire problem.

One problem we had was how to share work among the threads. Our solution was for the host to travel as far down the top of the decision tree as possible and statically divide up the work into as many small chunks as possible. Through experimentation, we found that we got the best results when the host traverses about 18 levels through the decision tree. Due to the unit propagation and pure literal substitution optimizations, significant portions of the search space are eliminated by the algorithm. By having the host divide the search space into as many sub-trees as possible, we keep each core busy because the GPU will simply assign another block to a core when it becomes idle. This approach also lowers the probability that the GPU is waiting a long time for a single core to finish when all the other core are idle.

For SAT problems that are satisfiable, if a thread finds an assignment that is satisfiable, all other cores can stop searching for a solution because a solution has already been found. To account for this case, we use an atomicAdd() operation to inform the other threads that a solution has been found. If a solution has been found, no new blocks should be scheduled to an idle core. This approach is a simple way of exiting earlier without harming performance on SAT problems that are unsatisfiable, which typically take longer to solve.

V. Results

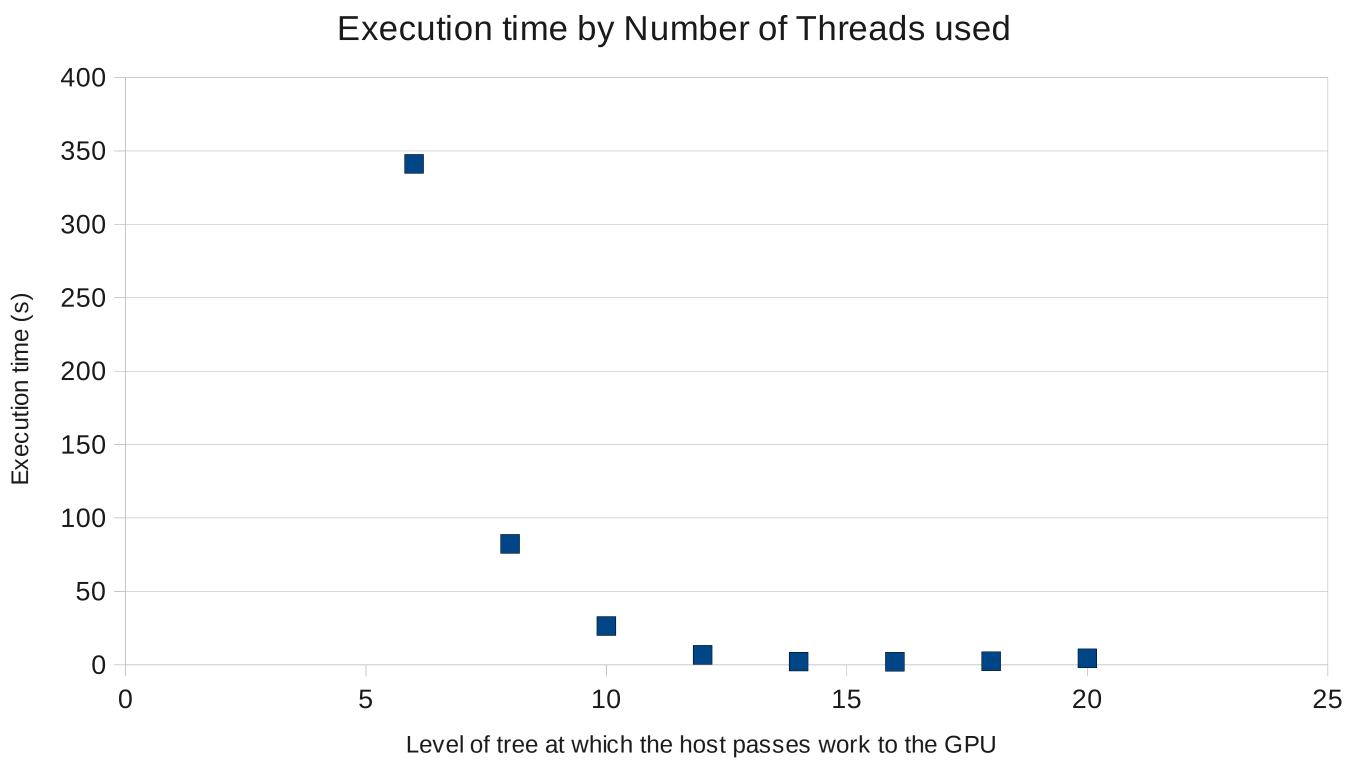

Figure 1: Execution time as the depth of the tree that the host traverses increases.

As we increase the number of partitions into which the work is divided into, the speed at which the GPU can solve the problem increases. For example, if the host traverses 5 levels down the tree, all work will be divided between 32 threads. As we can see from Figure 1, as we increase the number of levels traveled by the host, the execution time decreases. As the work is partitioned into smaller and smaller pieces, we achieve better work balance. The large drop in execution time between levels 6 to 8 occurs because there is not enough pieces of work for all cores to be busy.

Results for individual benchmark files (host partitions work on the 18-th level of the decision tree):

File	CPU (secs)	GPU (secs)	Speedup
dubois20.cnf	105.299	2.366	**(45.6x)**
dubois21.cnf	233.4	4.285 s	**(54.4x)**
dubois22.cnf	508.8	8.850	**(57.7x)**
uuf120-020.cnf	13.447	11.597	**(1.16x)**
uuf125-030.cnf	31.574	27.867	**(1.13x)**

When comparing the GPU and CPU times, we found that the GPU version compares well to the CPU version for hard SAT problems. A hard SAT problem means that the SAT problem contains few heuristics that can be used to prune the tree. A randomly generated SAT problem is generally considered a hard SAT problem. For problems where the pruning heuristics can be applied many times, the work balance can become unbalanced. We combat this behavior by dividing the decision into as many sub-trees as possible. However, for problems like uuf120-020 and uuf125-030, this way to balance work is less effective. Also, note that the problem size is important. For simple files (less than 20 variables and 50 clauses), the CPU and GPU time are nearly constant. It is only for larger problem sizes that we see significant speedup.

For some instances of SAT, pruning the tree can cause the sub-trees to become unbalanced such that one sub-tree is much larger than the rest of the sub-trees. This problem can occur even though the host attempts to split the work into a large number of small sub-trees as possible by traversing as many levels of the tree as possible. One approach that we tried to solve this problem was to detect when a core became idle because all remaining sub-trees were already being worked on by another core. When we detected that this was the case, a non-idle core would give part of its own work to the idle core. Due to time constraints, we did not finish implementing this idea, and future work would be to evenly balance workload in all situations without too much overhead. In this case, we hope that the overhead of synchronization would not negatively affect the benchmarks in which workload balance was not a problem.

Because of the greater computation power of the GPU to the CPU, the GPU is an attractive choice for solving SAT problems. We mostly achieved our goal of obtained a large speedup many SAT problems. Unfortunately, we did not achieve similar results for other SAT problems. However, in many cases, the GPU can outperform the CPU suggesting that in these cases, the GPU is a good choice over the CPU.

VI. How to run the programs

The code is located in the *serial/* folder and the parallel version in the *parallel*/ folder. The Makefile will generate an executable. Run the executable with any one of the test files in the *tests/* folder as the first and only argument. The output of the program will be an assignment of the literals if the formula is satisfiable or "no solution" if the formula is unsatisfiable.

Note: the input files are in DIMACS format. The format is widely accepted as the standard for boolean formulas in CNF. Essentially a line in this format specifies a clause. A positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero. More information on the DIMACS format can be found in the References section.

VII. References

Boolean Satisfiability (SAT) Problem
http://en.wikipedia.org/wiki/Boolean_satisfiability_problem

DPLL Algorithm Description
http://en.wikipedia.org/wiki/DPLL_algorithm

DIMACS input format
http://logic.pdmi.ras.ru/~basolver/dimacs.html

Benchmark files:
http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html
http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

VIII. Assignment of Work

Charles did most of the work implementing the CPU solver, while Michael did most of the work implementing the CUDA solver. More specifically,

Michael Wang -
Implemented and optimized CUDA version of algorithm
Implemented and optimized serial version of algorithm for CPU
Wrote up checkpoint, final report

Charles Guo -
Implemented and optimized serial version of algorithm for CPU
Wrote up proposal, checkpoint, final report
Maintained Website