Parallel SAT Solver using CUDA - Odell Blackmon III & Viraj Puri: Final Report

Summary

For our final project, we parallelized the Davis-Putnam-Logemann-Loveland (DPLL) SAT solver algorithm, using the resources of the NVIDIA CUDA platform on the CMU Gates GPU's, to speed up its performance compared to the popular serial version. We took several approaches to parallelizing the algorithm using CUDA, then compared their speedup on three different sized inputs against a baseline serial implementation of the DPLL algorithm. With these results, they were then all compared, contrasted, visualized, and analyzed for each of their performances.

Background

Our project centered around finding a faster version of the popular DPLL algorithm using NVIDIA's CUDA framework. Before describing the DPLL algorithm, some background problem context is necessary:

1. **Literal:** This is a variable (that is at its simplest form, also called an atom), that can be either a variable or its negation. **Examples:** A, ¬A, B, ¬B, etc.

2. **Clause:** This is a formula/collection of literals that are joined together by logical connectives (such as Λ (and), V (or)). **Examples:** (A V B), (¬ A V B Λ ¬ C), etc.
3. **Conjunctive Normal Form (CNF):** Simply put, CNF is a specific way of representing numerous clauses together (also called a formula), in which each clause is connected together using the logical connective Λ (and), and within each clause, each literal is connected together using V (or). **Examples:** A, (A V B), (A V B) Λ (¬ A V C), etc.
4. **Unit Clause/Literal:** A unit clause is a clause in which all but 1 literal have been assigned a value of 'False,' except 1 literal, A, and we call this literal a unit literal. This term is extremely important because in our DPLL algorithm we will often hunt for a unit clause and find the unit literal in it to assign a value of 'True' to make the whole clause True. **Examples:** (A V B V ¬ C) where A is unassigned, B is assigned 'False,' and C is assigned 'True' to make ¬ C have a value of 'False,' thus making A also be the unit literal and the whole clause is the unit clause.
5. **Pure Literal/Pure Literal Assignment:** If a literal appears as only one polarity in our entire formula, and never has its negation anywhere in our formula. This is especially helpful in the DPLL algorithm since if it only has 1 polarity we can set all instances of it to 1 truth value to make it true, so we no longer have to worry about its assignment. **Example:** (A V B V ¬ C) Λ (¬ B V C) Λ (B V ¬ C V A), here A is a pure literal since it only appears as A in each clause, having a positive polarity, thus we might as well assign A to be true in every clause. If it was ¬ A everywhere it would also be a pure literal with negative polarity, and we might as well assign A to be false in every instance so that ¬ A is true everywhere.

6. **Unit propagation:** If we have a unit clause, then this clause is only made true if we assign the unit literal to be true to make the clause true, so we don't have to make a random truth value assignment for a literal like we would have if we didn't have any unit clauses. So, with this, we remove every clause with a unit literal.

With these terms at our disposal, we can now discuss the details of the DPLL algorithm which we sought to optimize:

Input: X, a formula of logical expressions in CNF form

dpll(X):

1. While(unit clauses are present in X & there are > 1 clauses in X):
    a. Run unit propagation on X and a unit clause in X, C, and reassign X to be this.
2. While(pure literals are present in X):
    a. Run pure literal assignment on X and a pure literal in X, L, and reassign X to be this.
3. if(X is empty):
    a. Return 1.
4. if(X contains any empty clauses):
    a. Return 0
5. Set a ‘result’ variable = 0
6. Get a new literal L from our set of clauses and set result = dpll(X U L), a recursive call with X and our new literal L.

7. If the result of this is 0, i.e result == 0 after step 6, then try using the negation of L with X, i.e set result = dpll(X U $\neg$ L).
8. Return 'result.'

Output: 0 or 1, where 0 means formula is not satisfiable, and 1 means the formula is satisfiable.

Data Structures used:

So, we represented each variable as an integer from 1 to N, where N is the number of variables in our formula, where each number, i, in this range from 1 to N, would represent the variable x_i, and could be represented as -i as well, to represent $\neg x_i$. One line of these numbers would represent an entire clause, and all lines together would be our entire formula. So, for example:

If we had two lines as input as follows:

1 2

-2 3

This would represent the formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$.

This way of representing our inputs was inspired by and is close to the popular CNF form representation according to the popular DIMACS representation of CNF text input formulas, with a few subtle differences of representation.

So, with this as our input, we noticed that a formula is essentially a collection of separate clauses, where each clause is just a collection of variables/numbers.

With this, then, we decided a 2D representation would be necessary so in our C++ representation we modeled an input formula X using the following data structure:

X = Vector of clauses, C = <C_1, C_2, ...C_n>, n = # of clauses

C_i = Set of numbers/Variables {a_1, a_2, a_3, ... a_m}, where a_i = a specific # ranging from 1 -> # of variables in formula X, and m = # of variables in clause C_i.

The key operations on our input data structure X, as also previously described above in our algorithm, are that X is used in each of the main parts of our algorithm, to include running unit propagation, pure literal assignment, and then checking for empty clauses vs checking if X is empty as our base cases. It is also often modified as we set X to be the result of unit propagation on X in our first while loop, and set X to be the result of pure literal assignment on X and it's pure literal in the second while loop. Then we finally also re-rerun the recursive DPLL algorithm at the end using a new literal, L, from X and assigning it a value of true in X then recursively calling the dpll algorithm on this new X which has a new truth value for L in it. We also do the same for the negation of L to see which recursive result gives us back a satisfiable result.

Since we only have 1 data representation input into our entire algorithm, it makes intuitive sense above why it is used in various read/modify/write operations above, as it is our only data structure we have to work with. Thus, picking the correct data structure to represent our formula is essential in that it would affect our ease-of-implementation, potential runtime, and how easily we are able to map a serial version to a parallel one.

Our choice of representation was also carefully thought out:

Vector for collection of clauses: We chose to use vectors to represent clauses since they have the ability to resize when necessary, and accessing each clause is quite simple using iterators which is vital in our situation since this is done quite often. Also, the fact that they can dynamically change its size on its own is especially useful since in our recursive calls we often modify a clause with a new truth value for a specific literal.

Set for the variables in each clause: Sets in general are faster than a simple list/array representation since they are implemented using BST's, but they have requirements in that each element must be unique, in increasing order, and that a value in a set cannot be modified, but, instead one can remove that value and insert the modified one instead. These 3 conditions actually work perfectly for our representation of each clause, since each clause in our DIMACS-like representation is represented using integers in increasing order, we never have more than 1 copy of a variable in a clause, and if we do need to assign a new truth value to a variable, removing it then re-inserting it is no problem for us, as our set will even take care of putting it in the correct increasing order.

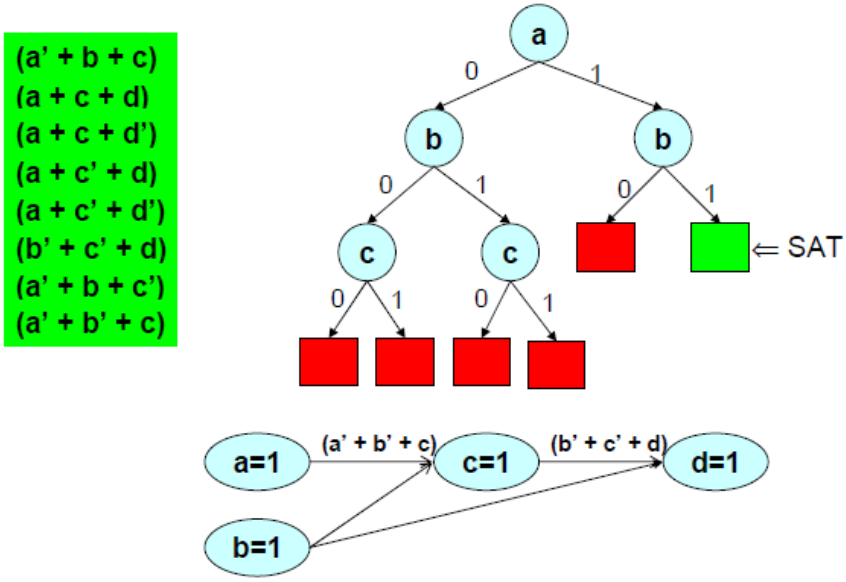
With the ideal data structures in mind, it is now worth considering which parts may be slow in an initial approach to the DPLL algorithm and how the workload/parallelization could be improved in such an algorithm.

Computation/Workload:

So, looking at the algorithm, we see a few potentially expensive portions:

1. First while loop where we run unit propagation
2. Second while loop where run pure literal assignment
3. The end of the algorithm, where two recursive calls are made which means that we have a decision-tree like structure in that each recursive call makes another two recursive calls and thus also calls both while loops, which could be very expensive.

In the worst case scenario, the DPLL algorithm backtracks when it reaches a base case that is unsatisfiable and then has to back track all the way up a ‘decision tree-like structure’ to where it made the last decision of picking a truth value and assigning a different value. This could lead to an exponential run time of $O(2^n)$, if every path down a branch has to backtrack.



Looking at a visual representation of such a backtracking scenario, we see that choosing a = 0 leads to several branches which fail, which would cause the algorithm to constantly backtrack no matter what we assign b or c to, and we only succeed after we assign a = 1, and b = 1, using unit propagation.

Thus, we see here that this is what our expensive operation is centered around, as especially with recursive calls constantly running the while loop, splitting to make a decision, then backtracking to run it all again, we see that our goal is thus to minimize the amount of backtracking that occurs.

The workload we are working with in this particular problem is the search space for all 2^n possible variable assignments that might make the expression satisfiable. The DPLL algorithm in particular is dependent on the prior manipulations done on the clauses to reduce the search space. The data itself is not parallelizable, but manipulating the clauses can be done in

parallel. There is not much locality to exploit for the algorithm since it is difficult to predict how and which clauses will be manipulated at different rounds of the algorithm. SIMD execution does not work the best for this particular type of problem since it is possible for there to be very little overlap between clauses.

However, we see that there is not much dependency in our program, as dependency is generally between recursive calls, as we need the result of a parent node (as shown above) which includes a variable assignment to our next recursive call in our stack. Also, looking at the serial algorithm, we see that not much parallelism currently exists, and that it is not data-parallel, as unlike most cases of the NVIDIA CUDA implementations, which rely on locality of data and variables, using constructs such as dividing up for-loops between threads/blocks, our task is significantly more challenging here.

Approach

Technologies Used:

We used the following technologies/languages:

1. **C++:** We used C++ as our main language for coding the serial implementation of the DPLL algorithm, as well as the fact that the NVIDIA CUDA framework uses mostly C++ implementation syntax, except for a few CUDA specific nuances. So, we ended up using C++ as the majority of our code for the serial version of the DPLL algorithm, our first approach at a CUDA-parallel implementation of the DPLL algorithm (the one that

included a nearly 1:1 mapping of our serial DPLL to a parallel one that retained the recursion), our CUDA-parallel implementation of a SAT solver that used a brute-force approach, as well as our final parallel implementation that used the iterative DPLL version optimized over the pure literal assignment loop and unit propagation loop.

2. **NVIDIA CUDA framework/GHC Machines:** We coded up our three parallel implementations (as described above) on the CMU GHC machines which use the NVIDIA GeForce RT 2080 B GPUs. These GPUs support CUDA compute capability 7.5, and was the basis for our experimentation for our CUDA implementations. We used the CUDA C++ framework for the actual programming implementation of our solutions, using concepts such as threads, blocks, kernels, device/host functions accordingly.
3. **Python:** We also used Python 3 to create a script that takes user I/O and after asking a user how many variables and clauses they would like, as well as if they want a satisfiable or unsatisfiable input, will generate a text file in the correct format of an input that our algorithm could take in and attempt to solve, as well as display how long it took (in seconds). The reason for this is Python was not only a language that our group members were familiar with, thus causing the creation of this script to not take up too much time, but it was also very well suited for string manipulation which was the core of the creation of the .txt file that was created as a result of this script. This way, we could test various sizes to create an “easy,” “medium,” and “hard” sized input file to have various measures to test our solutions on, similar to prior 15-418 assignments.

Versions of Algorithm/Timeline:

We went through many iterations of a SAT solver to identify what would be the most 'ideal' one. Our steps of optimization went as follows (in chronological order):

1. **Serial Implementation:** Starting with the outline of the DPLL algorithm found on Wikipedia (linked in the References section of this report), we first implemented a serial, recursive implementation of the DPLL algorithm to get baseline results on how it fared on inputs of various sizes. After implementing the serial version and getting it working, we tested it on small test cases (involving variable sizes from 10 - 100, about 1.5x # of variables as the number of clauses) and got decently promising results. Since this was a starting point and used no parallelization/optimization, we tested how long it would take on various sizes, as well as identified what caused the biggest change in time increase for a larger input by running the serial implementation on an input with more variables, while holding the number of clauses constant, and vice-versa to identify the bottleneck factor. Through this, we found that increasing the number of variables increased the time to execute the DPLL algorithm heavily, while the number of clauses was not really a deterring factor. This makes logical sense, as looking at the decision tree above, we would have to run our algorithm many more times for larger variables, due to the expansion of the number of nodes in the decision tree. So, we found that the backtracking bottleneck stemmed from the number of variables, and that the

worst-case $O(2^n)$ run time came from the number of variables. With these baseline results, we decided to make a first pass at a parallel solution.

2. **Parallel brute force approach:** We decided to use a brute force approach using the NVIDIA CUDA framework to see if using a parallel approach, but with a different algorithm would fare any better than a serial implementation. Here, we tried using the NVIDIA CUDA framework and created a kernel designed for variable assignment. The brute force approach is literally as it sounds as it tries every possible assignment of variable values until it reaches a valid one. Unfortunately for this, we found that for each additional # of variables (i.e going from 5 as the number of variables to 6 as the number of variables), would significantly hit the compute time, until about 30 variables when the brute force approach would not work anymore (as it would segfault). The reason for this is because the brute force approach has to try about $2^{\#\text{ of variables}}$ different ways in the worst case to assign every possible value and combination to every variable of True/False, and thus, our representation of the array of integers space bound is likely reached, and thus overflows on the CUDA memory, causing a segfault. With this in mind, we
3. **Parallel DPLL algorithm (Version 1):** After noticing that the brute force approach, even though it was parallelized, took far too long to process a significant number of variables, we decided that now we could port the serial version of the DPLL algorithm to a parallel implementation. We attempted this in an effort to find if we could improve upon our two prior versions which took far too long, since the serial version only used 1 thread to process everything, whereas the brute force approach, even though it used

multiple threads, each thread was using a brute-force approach, which together with many threads would take far too long. In doing so, we had several constraints to work around. First, implementing recursion on NVIDIA CUDA. This proved to be especially challenging as although the CUDA kernel is where one should be doing much of the processing, kernels must return a void type, and cannot use any recursive calls to itself. Second, was trying to figure out how to convert two while loops to work in parallel, as CUDA generally is assisted in locality of data, and being able to work on separate, independent, usually contiguous blocks of data (like nested for loops), but here we did not have such a scenario, presenting an additional challenge. In the end, for this V1 parallel DPLL implementation, we decided to use it as a baseline solution for a first try at parallelizing the DPLL algorithm and ended up using a host function which would essentially call the serial implementation of the DPLL algorithm from 1 thread/block to handle everything in a __device__ function that is called from the kernel which is called from the host since recursion cannot be implemented in a kernel, and parallelizing recursion is very difficult.

4. **Parallel DPLL algorithm (Version 2):** From our V1 parallel DPLL implementation we noticed several issues which led us to create a V2 version of the parallel DPLL algorithm. 1. Implementing recursion on CUDA is not ideal, as the memory of a GPU is far more limited than that of a CPU, so when a recursive call is made, even from a __device__ function which supports recursive calls, the limited memory leads to a much more limited recursive stack memory available. This led our V1 parallel version to not work then for any decently large test case. 2. We were not optimizing any of our

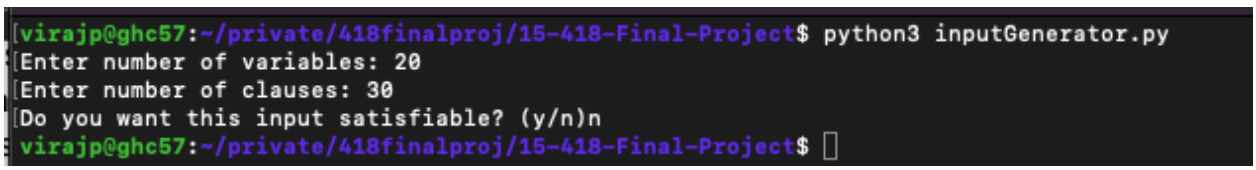
bottlenecks. In our first iteration of the parallel DPLL algorithm, we never optimized over our most expensive operations, taking advantage of the numerous threads available to us to see if we could speed up our expensive backtracking algorithm at all. Thus, with these two reasons together, after implementing the V1 parallel DPLL algorithm, we had to abandon it in favor of an advanced V2 version. This V2 version improved upon the V1 parallel DPLL algorithm by leaving parallelism to the CPU and parallelizable aspects of the algorithm to the GPU. The CPU is simply better equipped to handle deep recursive stacks so we chose to leave those aspects of the algorithm to the CPU. However, we noticed that there are certain manipulations of the clauses that can be done in parallel such as unit propagation and pure literal elimination. We relegated these operations to kernels and assigned each thread to a clause to manipulate.

Optimizations/Problems/Mappings:

We ended up not changing our original serial algorithm to better map to a parallel machine. The reasoning for this is that our original serial DPLL algorithm already performed decently well (as will be seen in the Results section), and that in order to map to a parallel machine, we couldn't just merely change the way we stored values in our data structures. We had to become much more creative and find a way to implement the recursive algorithm in our CUDA V2 version by taking advantage of the various threads, and figuring out how to use two different kernels for each of the pure literal assignment and unit propagation loops. Thus, it was not necessary to change the original serial implementation, since we could just get creative and use new changes in the parallel version.

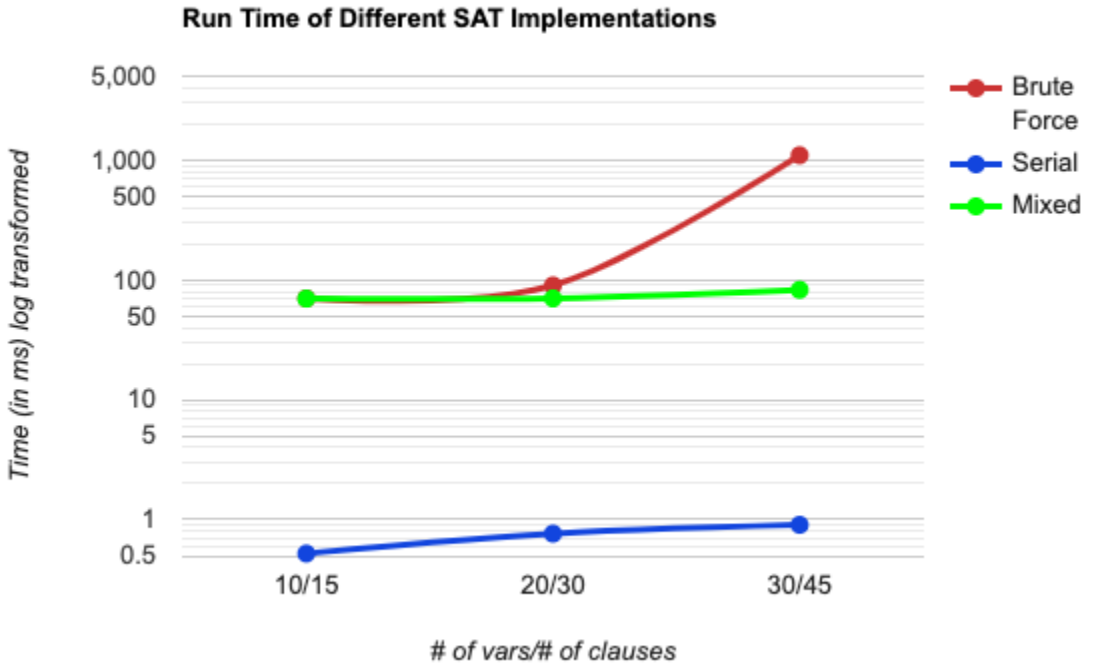
Our first attempt at parallelism required us to abandon our initial representation of the problem to accommodate for the limitations of using CUDA. Our initial representation involved using a vector of sets of integers to represent the different clauses we will be manipulating. This was perfect for our algorithm since vectors and sets afforded us constant time lookup for information such as whether a variable is in a clause or how many elements are in a clause. CUDA does not allow for such data structures so we had to work around this by flattening our original data structure into three one dimensional arrays. The first array stored all of the clauses back to back with integers representing a variable and whether or not it was negated. The second array stored the starting indices for each clause in the first array. And the third array stored clause length information. We then used all three of these arrays to divide the work of manipulating the clauses across threads. Each thread was assigned a clause and then performed either unit propagation or pure literal assignment. Work could not be subdivided further to focus on individual variables since how a particular variable should be manipulated depends on its neighbors.

Results



So, we had three versions of our parallel implementations that we could reasonably test, since the other parallel version would fail on a decent sized input. Unfortunately, the brute force approach did not support input variable sizes of > 30 due to lack of space representation. So, we decided to set up an experiment to measure the time it took for the three versions, serial,

brute-force parallel, optimized dpll parallel. We set this up by using our random input generator (I/O shown above) which took in a specific number of variables/clauses and created text files for us to use in each case. With this, we had 3 ‘levels’ of difficulty to test our approaches on: 1. **Easy:** 10 variables, 15 clauses. 2. **Medium:** 20 variables, 30 clauses. 3. **Hard:** 30 variables, 45 clauses. We decided to vary the number of variables, but kept the number of clauses as 1.5x the number of variables for each.



Looking at the graph above, we actually took the y-axis values, which was the raw execution time (in milliseconds), and performed a log-transformation on it since the scaling of execution time was vastly different between the serial and the brute force as the brute force approach ended up taking 1000s of milliseconds in the worst case, whereas the serial version only took

about 0.8-1. So, to keep the lines all on one graph and keep the trend of the lines still visible we performed a log transformation on the time in ms.

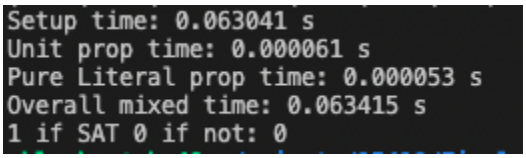
Analysis of results:

Looking at the results of our experiment, we see that unfortunately, our mixed approach did not perform the best in terms of raw time performance. The serial version ended up performing the best as for a large number of variables, it only took about 1 ms. On the other hand, in the worst case, the brute force approach took about 1096 ms, and the mixed approach took about 82 ms. So, the mixed approach did perform about 13x better than the raw time for the brute force approach, which is to be expected as it utilized its numerous threads, as well as implemented the DPLL algorithm vs. not utilizing the threads and using a brute force algorithm. However, it is important to note that in terms of scalability, the serial version ended up almost doubling in time from the smallest number of variables to the largest input size, whereas the mixed approach only took about 13% more time from the smallest to largest input size. This is very interesting to note as in terms of scalability, our mixed approach scaled the best in terms of raw time. These results are indeed way better than the brute force results which took > 12x more time from the smallest to largest input size.

While we did intend to initially use the serial version as a baseline reference and then improve upon it, we ended up getting evidence that in this case, when measuring performance in terms of raw execution time the serial version performed the best. When measuring performance in terms of scalability, the mixed version performed the best. The lack of overhead of the single core serial version also helped make the raw run time of the serial version the best.

What limited our parallelism the most would have to be the setup overhead. We expected our mixed parallel approach to lead to some speed up on the unit propagation and pure literal

assignment steps, but these gains in speedup were overshadowed by how much time was spent moving data between the CPU and the GPU. For the mixed parallel approach to work, it required a significant number of calls to cudaMemcpy and cudaMalloc as well as allocating different sized arrays to handle how the clauses evolve over time. For instance, on one of our inputs, setting up the cudaArrays took up close to 99% of the time as seen below:



Overall, we believe that a CPU would have actually been a better choice of implementation. Our whole motivation behind this project/experiment was to see the results of combining GPU processing ability for a scenario that did not seem to be optimal for a GPU, and our doubts that the fastest serial version would be better than a very fast parallel version were confirmed. This stems from the fact that GPUs/CUDA performs best for large floating-point operations, large numbers of math operations, scenarios that don't require dependencies. Additionally, CUDA is ideal for large numbers of tasks that require identical/near-identical operations. They struggle, however, for situations that require multitasking, and extremely complex operations such as logic branches, which is more or less the core of the DPLL algorithm. On the other hand, our DPLL algorithm had minimal math operations, rather its essentially a 'smart' trial-and-error algorithm, and there was a large number of dependencies as each successive call depended on a prior calls assignment of truth values to the formula. The DPLL algorithm also depended on almost entirely logical statements (if statements and while loops) and then branching out to run another complex task, thus posing an almost completely unideal circumstance for CUDA.

References

1. **DPLL algorithm**: https://en.wikipedia.org/wiki/DPLL_algorithm
2. **NVIDIA CUDA reference guide:** https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
3. **C++ reference guide:** https://www.cplusplus.com/reference/
4. **CNF DIMACS representation:** https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/SATLINK____DIMACS
5. **Line Graph Maker:** https://www.rapidtables.com/tools/line-graph.html
6. **Information on CPU vs GPU:** https://www.weka.io/blog/cpu-vs-gpu/

Distribution of Work/Credit

Distribution: 50-50%

List of work per student:

Odell:

1. Serial implementation
2. Parallel brute force implementation
3. Parallel DPLL optimized over pure literals/unit propagation
4. Final project presentation

Viraj:

1. Created a Python script that generated input files as well as ran extensive trial and error to find ideal input files for each of our implementations.
2. Created first attempt at the CUDA parallel implementation of the DPLL algorithm which used recursion

3. Final Project Report/Graphs
4. Final Project presentation slides/presentation