15-418 F22: Parallel Computer Architecture and Programming
Term Project: CPU and GPU Accelerated Parallel SAT Solver

Name: Jiarui Wang, Tejas Badgujar
Andrew ID: jiarui wa, tbadguja

Project Github URL

1 Summary

We will implement a parallel boolean satisfiability problem solver. After implementing the solver with thread-level parallelism on CPU and CUDA parallelism on GPU, we will run our program on the GHC machine (or another machine similar in specs) to measure and compare the performance and speedup of our algorithms.

2 Background

As the first problem proven to be NP complete, SAT is undoubtedly one of the most iconic and fundamental computation problem in computer science. It tightly connects to many problems such as type checking, automated theorem proving, automated circuit verifying and many other fields.

Boolean satisfiability problem can be summarized into the follow sentence: given a boolean formula, is there an assignment of variables that will evaluate the boolean formula to true.

To add more detail, a boolean formula is composed of some very basic elements. There are three logical operators $\neg$(not), $\vee$(or) and $\wedge$(and) that connect variables together. These logical operator follows the following basic rules

x	$\neg x$
true	false
false	true

x	y	$x \vee y$
true	true	true
true	false	true
false	true	true
false	false	false

x	y	$x \wedge y$
true	true	true
true	false	false
false	true	false
false	false	false

Parenthesis could be used to change the order of evaluation just like arithmetic expressions. Since the SAT problem is proven to be NP complete, in the worst case scenario, we would need to evaluate all possible assignment of variables to decide the satisfiability of the boolean expression. The number of possible assignments grows exponentially with the amount of variables since every variable have two possible assignments, true or false.

Note that this exhaustive search across an exponential amount of assignments would benefit greatly from parallelism since the evaluation of each assignment is completely independent from each other and each worker could be assigned with a certain segments of the search and evaluate different assignments themselves. Additionally, other approaches to SAT solving, such as those using DPLL logic and conflict-driven solving, offer speedup compared to the naive implementation and could possibly even further improve from a parallel environment.

Since we could use boolean algebra to convert any arbitrary boolean formula into conjunctive normal form, our program will only solve SAT problems in conjunctive normal form. If we have time, we could write a converter that will automatically convert any boolean formula into CNF.

A conjunctive normal form is where a boolean formula is only consist of conjunctive of clauses. A clause is a disjunction of literals. Literals are either a variable or the negation of a variable.

To put it more concretely, x is a variable that could be assigned true or false value. Therefore, the following are both literals:

$$x, \neg x$$

Similarly, x, $\neg x$ and y are literals and if we connect them together with $\vee$ to form a disjunction, we have a clause:

$$x \vee \neg x \vee y$$

To put everything together, if we connect the clauses together with $\wedge$ to form a conjunction of disjunctions, we have a boolean formula that is in conjunctive normal form:

$$(x \vee \neg x \vee y) \wedge (z \vee \neg y \vee x) \wedge (\neg x \vee \neg z \vee \neg y)$$

Note that the CNF above is satisfiable with the assignment $x = true, y = false, z = false$ since if we plug in the assignment into the CNF, we have:

$$\begin{aligned}
&(x \vee \neg x \vee y) \wedge (z \vee \neg y \vee x) \wedge (\neg x \vee \neg z \vee \neg y) \\
\Longleftrightarrow &(T \vee \neg T \vee F) \wedge (F \vee \neg F \vee T) \wedge (\neg T \vee \neg F \vee \neg F) \\
\Longleftrightarrow &(T \vee F \vee F) \wedge (F \vee T \vee T) \wedge (F \vee T \vee T) \\
\Longleftrightarrow &T \wedge T \wedge T \\
\Longleftrightarrow &T
\end{aligned}$$

3 The Challenge

- **Workload:**
  In the naive implementation, the entire problem requires exhaustive search through an exponentially large number of variable assignments. The entire search space is going to be evenly distributed to the workers and every worker should have a local copy of the boolean formula and evaluate variable assignments that are assigned to them.

  There is no dependency between the evaluation of variable assignments so there should be minimum communication between workers and it will be a highly computation intensive workload.

  Nevertheless, the evaluation of variable assignments will have drastically different workload depending on the assignment. For example, for a CNF problem with one million clauses, some assignment might cause the first clause to be false and the worker don't have to evaluate further because for a CNF problem to evaluate to true, every single clause in the CNF will have to be true. Meanwhile, some other assignment might fail on the last clause.

  In other word, some assignment may only require the worker to go through a very small subset of the CNF while other assignment may require the worker to go through the entire CNF. This will make workload balancing very tricky.

  There will be high spatial and temporal locality since the worker will refer to the same variable

assignment various times during evaluation and the worker is highly likely to evaluate the next adjacent clause in the CNF.

However, after performing some research, there are certain parallel-specific algorithms for SAT solving that we would also like to explore and implement. One such approach is the portfolio approach: instead of splitting the search space amongst processors, it chooses to rather assign each processor the entire search space but have each processor use a different algorithm. Furthermore, with such an approach, we could also look at a mixed approach; with a sufficient amount of cores/nodes, we could group sets of nodes together to all use the same algorithm and also divide up the search space amongst the group. Again, such an approach would not have a lot of communication between processors and would also have high spatial and temporal locality.

Furthermore, we also plan on exploring on whether or not using a GPU may prove beneficial in terms of SAT solving. Our intial assumption may be that the naive divide-and-conquer approach can benefit from the high levels of parallelism a GPU can offer but the size of the problem itself may be too large to fit for a GPU unit. The results of this exploration will be presented in future writeups.

We could also explore the portfolio approach in GPU with each warp group performing a different algorithm and using divide and conquer algorithm within each warp group. But that will also require further research on the GPU architecture and SAT solving algorithms.

- **Constraint:**
  Note that since GPU has limited local memory per warp block, we probably would not be able to load the entire CNF problem into local block memory. We need to figure out a good way to break up CNF into chunks of clauses.

  On the other hand, we need to figure out how to map variable assignment or different algorithms to each CPU or GPU thread. Mapping to different CPU core should be easier since every core has their individual instruction stream. But mapping to different GPU thread will be challenging since CUDA programming achieves its high parallelism with SIMD and the portfolio approach would require different instruction stream since different thread is supposed to use different algorithms.

4 Resources

We will start from scratch for our implementation. As of currently, we are using Wikipedia as a general reference to the boolean satisfiability problem[1] and SAT solver[2].

We will be using GHC machines to develop and test our code since we have easy access to them and they have good CPU and GPU to test out the performance of our code.

5 Goals and Deliverables

5.1 Separate Goals

- Our planned goal is to implement a strong parallel SAT solver with around linear speedup (if only using a CPU-based approach) in the amount of cores that we use compared to the naive implementation or with strong parallelization (if using a GPU-based approach) compared to the naive implementation. We expect to get 90% (A) if we are able to achieve the following goals.
  - Implement a robust naive single-threaded solution on CPU that will use brute force to search through the exponential amount of possible assignments.
  - Implement a thread level parallel solution on CPU that uses divide and conquer to search through the exponential amount of possible assignments.

- Implement a CUDA parallel solution on GPU that uses divide and conquer to search through the exponential amount of possible assignments.
- Implement strong sequential, single-core algorithms (DPLL and CPCL algorithms) on CPU for SAT solving.
- Combine DPLL and CPCL together to implement a portfolio based parallel SAT solver on CPU.

We are not yet sure about how much speedup we may get with something like a portfolio or mixed approach. We may have more accurate numbers to publish after intial exploration.

- If possible, after these implementations, we may research into more complex single-threaded SAT solvers and see if we can find a ideal balance between number/type of algorithms, number of cores, and problem size to find the best fit for any SAT solver.
- If we have enough time, we might also look into implementing a portfolio based approach on GPU and see if we could combine the SIMD nature of each warp block and how each warp block could have different instruction streams to create a portfolio and divide and conquer hybrid solution in CUDA.
- Additionally, if we are able to find them, we hope to be able to test our algorithm on a set of inputs comparable to that of the "International SAT Solver Competition" and see how we perform compared to other SAT solvers. However, this may prove difficult, as the test inputs are likely private and we would somehow need to make sure our machine types/specs are similar to those that other benchmarks/competitors are running on.

5.2 Demo Plan

We will be demonstrating speedup graph of our many parallel algorithms during the demo. If we are able to achieve linear speedup (good workload balance in divide and conquer) or even super linear speedup (use of better algorithm), we have done a good job implementing these parallel algorithms.

6 Platform Choice

We will use the GHC machine to develop and debug the code. Since GHC machine has a high performance CPU (i7-9700 with 8 core count) and a high performance GPU (RTX 2080), it will be great for us to test out the speedup of our CPU and GPU parallel code. Both of the team member also have easy access to the machine which makes testing and developing much easier.

However, we may choose to run on a local Windows machine that one of our groupmates has (which has a i7-9700K and a RTX 3080) for ease of access and possibly use the PSC Regular-Memory machines based on the results of our intial exploration.

7 Schedule

- Week 1 (11/7-11/13) Finish writing project proposal and setting up development environment
- Week 2 (11/14-11/20) Finish writing the sequential versions (naive, DPLL and CPCL) of SAT Solver that will run on CPU. Parallelize the SAT solver on CPU with thread-level parallelism with naive divide and conquer algorithm.
- Week 3 (11/21-11/27) Start writing the CUDA divide and conquer implementation of the SAT Solver. Combine DPLL and CPCL together to implement the portfolio version of the parallel code on CPU.
- Week 4 (11/28-12/4) Finish implementing CUDA version of the SAT Solver. Finish the portfolio implementation on CPU.

- Week 5 (12/5-12/11) Finalize the code. Measure performance and write up final report.
- Week 6 (12/12-12/18) Final Presentation

References

[1] "Boolean Satisfiability Problem." Wikipedia. Wikimedia Foundation, October 24, 2022. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[2] "SAT Solver." Wikipedia. Wikimedia Foundation, September 1, 2022. https://en.wikipedia.org/wiki/SAT_solver.