Parallel DPLL for Multi-Core CPUs and NVIDIA GPUs

Benjamin Colby, Griffin Teller

April 29, 2025

1 Summary

We developed parallel implementations of the Davis–Putnam–Logemann–Loveland (DPLL) SAT-solving algorithm for multi-core CPUs and NVIDIA GPUs. This report provides background on SAT-solving and DPLL, explores our development and optimization journey for both implementations, and compares the performance of each.

2 Background

2.1 DPLL

All efficiently verifiable computational problems can be transformed efficiently to an instance of CNF-SAT: given a conjunction of disjunctive clauses, does a satisfying assignment exist?

DPLL is a SAT-solving algorithm on which most modern solvers are based. The DPLL algorithm solves CNF-SAT problems by assigning literals, identifying unit clauses and pure literals, and backtracking after reaching a contradiction. The pseudocode for DPLL is described below [2]:

```
Algorithm DPLL
    Input: A set of clauses F.
    Output: A truth value indicating whether F is satisfiable.

function DPLL(F)
    // unit propagation:
    while there is a unit clause {l} in F do
        F ← unit-propagate(l, F);

    // pure literal elimination:
    while there is a literal l that occurs pure in F do
        F ← pure-literal-assign(l, F);

    // stopping conditions:
```

```
if F is empty then
    return true;
if F contains an empty clause then
    return false;

// DPLL procedure:
l ← choose-literal(F);
return DPLL(F & {l}) or DPLL(F & {¬l});
```

1. *Unit propagation:* Whenever a clause has exactly one unassigned literal (a unit clause), that literal must be assigned `true` to satisfy the clause. Propagating these forced assignments may produce new unit clauses.

2. *Pure-literal elimination:* A literal appearing only with one polarity (always positive or always negated) can be assigned to make all those clauses `true`.

The DPLL algorithm exposes two potential modes of parallelism:

1. *Data-parallelism:* Both unit propagation and pure-literal elimination iterate over clauses, and can be parallelized by partitioning the clause set among threads.

2. *Fork-join parallelism:* In the outer branching step, the two recursive calls are independent subproblems and can be forked off to worker threads.

2.2 3-SAT

3-SAT is the restriction of the general Boolean SAT problem to formulas in conjunctive normal form with at most three literals per clause. Determining the satisfiability of a 3-SAT formula is NP-complete, so it captures SAT's full worst-case difficulty [1]. Because many real-world decision and optimization problems naturally reduce to 3-SAT and there are many publicly available benchmarks, we chose 3-SAT as the benchmark for our implementations. Specifically, we chose the SATLIB 3-SAT benchmark suite for testing [4].

3 Approach

In this section, we explore our optimization journey in depth. Our final implementations are those described in 3.1.2 and 3.2.3.

3.1 CPU

The CPU implementation is written in C++, and the primary data structures are the formula and assignment. The formula is represented as a contiguous list of clauses, where each clause is represented as a contiguous list of literal IDs (ints). The assignment is represented as a contiguous list of `true`/`false`/`unassigned` values, indexed by literal IDs. Testing was performed on an Intel Core i7-9700 without hyperthreading.

3.1.1 Iteration 1: Naive Fork-join

Our first attempt at a multi-core algorithm was a naive fork-join implementation, where child threads are assigned a portion of the search tree to explore. Child threads are spawned when the algorithm branches. They explore their sub-trees, and, when terminated (reached SAT or UNSAT leaf), report back to the parent thread. The pseudocode for this implementation is described below:

```
function DPLL(F)
    unit propagation
    pure literal elimination
    stopping conditions

    l ← choose-literal(F);

    if depth < MAX_SPAWN_DEPTH then:
        spawn pthread(do DPLL(F & {l});
        if DPLL(F & {¬l}) is SAT then:
            cancel and reap pthread
        else:
            wait for pthread to finish
        return pthread.result;

    else:
        return DPLL(F & {l}) or DPLL(F & {¬l});
```

We discovered two problems with this implementation that informed our future multi-core implementations.

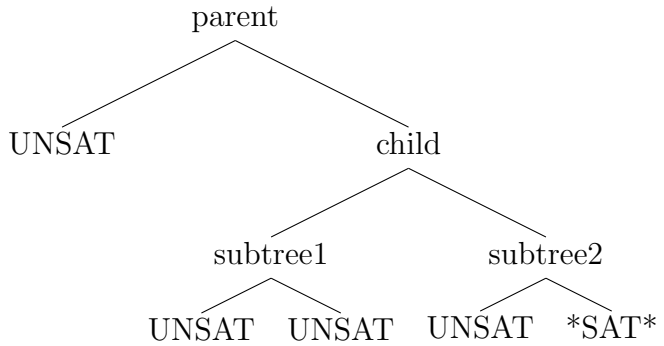
First, mutating and copying the formula is expensive. The DPLL algorithm mutates the formula during the unit propagation step, removing satisfied clauses so that we only have to search through unsatisfied clauses in sub-trees. While this makes traversing the formula more efficient, gprof indicates this triggers many expensive `realloc` calls (when threads mutate their formula copy) and `alloc` calls (when the parent thread creates a formula copy for their child).

```
index % time    self  children    called     name
<<< OMITTED >>>
-----------------------------------------------
[2]     36.9    0.01    0.10     218         vector<vector<int>>::push_back(...) [2]
<<< OMITTED >>>
-----------------------------------------------
[3]     36.5    0.01    0.10     654         vector<int>::push_back(...) [3]
<<< OMITTED >>>
-----------------------------------------------
[4]     31.3    0.02    0.07  302228         vector<int>::_M_realloc_insert(...) [4]
```

```
<<< OMITTED >>>
-----------------------------------------------
[5]     20.7    0.01    0.05  142477         allocator_traits::construct(...) [5]
<<< OMITTED >>>
-----------------------------------------------
<<< OMITTED >>>
```

Given that the formula copying/mutating accounts for a significant portion of execution time and it scales with the number of threads, the implementation stands to benefit from maintaining a single, immutable copy of the formula while maintaining smaller, mutable partial assignments.

Additionally, naive fork-join causes each parent thread to assume the worst-case performance of its children. When the algorithm branches, it runs the negative branch locally and gives the positive branch to the child to run. If the local branch is `UNSAT`, the parent thread sleeps until the child has finished; only then is it able to awaken and return to the grandparent. This means the parent is bounded by depth of the child's subtree, and thus the latency of every recursive call is the time to traverse deepest subtree of that node.



Consider the example above, the parent runs its branch locally and quickly hits an `UNSAT` leaf. It then sleeps until the child has fully explored its deep subtree and finally returns `SAT`. Implementing a way to return early when `SAT` is discovered could speed the implementation up significantly.

3.1.2 Iteration 2: Thread Pool

Our second (and final) implementation of the multi-core algorithm centers around a LIFO worker thread pool. When the DPLL algorithm branches, two tasks are pushed onto the thread pool task stack (one for the positive branch, one for the negative branch). A task consists of a single iteration of the DPLL algorithm. Idle worker threads grab tasks from the stack and execute them. If a worker thread ever encounters a `SAT` literal assignment, it signals the master thread, which immediately stops all workers, reaps them, and returns the satisfying assignment. If the stack of tasks every empties and no tasks are currently in flight, that indicates that the formula is `UNSAT`. The pseudocode for this implementation is described below:

```
function DPLL_master(F)
```

```
    create thread pool
    submit the first task to the pool

    wait for sat found cond signal or no tasks in flight

    destroy thread pool
    return satisfying assignment if found

function DPLL_child(F, A)
    stopping condition
    unit propagation
    pure literal elimination

    submit positive branch task
    submit negative branch task
```

As mentioned above, the thread pool is implemented as a centralized stack of tasks, which workers take from when they have no work. The pool has a single lock, which workers acquire to push/pop tasks to/from the stack, and to update counters about the number of tasks active/waiting. When tasks are pushed or popped from the queue, a single worker is woken up to take work. This reduces the pool lock contention even with a large number of workers waiting for work. The pseudocode for the pool (and its workers) is shown below:

```
function threadPoolSubmit(task)
    acquire pool lock

    // critical section start
    push task onto the stack
    // critical section end

    release pool lock
    signal to wake up one worker

function threadPoolWorker
    while true:
        acquire pool lock

        while stack is empty:
            release the pool lock and sleep

        // critical section start
        pop task from stack
        // critical section end

        release pool lock
```

```
        signal to wake up one worker

        execute task
```

We initially implemented the pool as a FIFO queue of tasks, but found that this had significantly worse performance than a stack. The algorithmic distinction here is the traversal method: DFS (stack) vs BFS (queue). Given multiple leaves are likely `SAT`, and there is some minimum depth that must be explored before any `SAT` nodes are found, DFS performs better. By pushing and popping tasks in LIFO order, the algorithm explores one branch to that critical depth, often finding a `SAT` node early, rather than interleaving work across all shallow nodes. In contrast, BFS keeps a large frontier of sibling tasks in the queue, which delays reaching the deeper subproblems where `SAT` solutions live.

3.2 GPU

The GPU implementation is written in CUDA, and the primary data structures are the formula and assignment. The formula is represented as a contiguous list of literal IDs (ints), which is understood by two other lists: a list of clause pointers and a list of clause lengths. The assignment is represented as a contiguous list of `true/false/unassigned` values, indexed by literal IDs. Tests were performed on an NVIDIA RTX 2080.

3.2.1 Iteration 1: Inner loop parallelism

Our first attempt at exploiting the data-parallel framework of GPU execution was to only parallelize the inherently data parallel inner steps of DPLL: unit propagation and pure-literal elimination. On the host side, our outer took the following form:

```
copy formula and assignment to device

DPLL(partial assignment):
    do
        anything propagated ← launch unit propagation kernel
    while anything propagated

    launch pure literal elimination kernel

    copy assignment from device memory

    if any false clauses:
        return UNSAT

    if all clauses satisfied:
        return SAT

    recurse decisions
```

The unit propagation kernel parallelizes over clauses, with each thread counting the number of assigned and true literals in its clause and setting the assignment of unit literals to true. All threads write to a bool in global memory `anyPropagated` if they any unit literal was propagated. The pure literal elimination kernel parallelizes over literals, with each thread looping through each clause to determine whether the literal only appears with one polarity. If it does, the assignment is updated.

Using timing instrumentation within our code, we discovered that a significant amount of time was spent eliminating pure literals (upwards of 90% on certain test cases). Knowing that sequential DPLL obtains the majority of its formula simplification from unit propagation [3], we decided to omit pure literal elimination from our GPU implementation. After disabling pure literal elimination, we observed substantially improved performance:

To find optimization opportunities, we started with where we suspected we still might leaving performance on the table: GPU utilization and kernel launch overhead. We suspected these might be an issue because this algorithm involves many launches to short-lived kernels, and the number of threads being assigned to those kernels was very low (equal to the number of clauses, which is under 1000 for all of our benchmarks). We were able to measure relevant utilization figures with NSight Compute, confirming that we would need to exploit the parallelism between decisions if we wanted to fully utilize the GPU.

Compute Utilization (%)	0.17%
Memory Utilization (%)	2.56%
Achieved Occupancy (%)	20%

The achieved occupancy refers to the occupancy *of occupied SMs*. In reality, we only occupy 2/46 SMs present on the RTX 2080.

Measuring kernel launch overhead was trickier. NVIDIA claims that kernel launch overhead can be expected to be near 5μs per launch, but to verify we compared the kernel durations in NSight compute with our own measurements from the host side. We observed kernel durations of $3\mu s$ to $6\mu s$ in NSight Compute, but we observed kernel durations of $10\mu s$ to $15\mu s$ on the host side. This implies a launch overhead on a similar order to the kernel duration, i.e, that we were speding nearly 50% of our kernel time just waiting for a launch. This indicated that we needed to schedule more work per kernel, which would also help with utilization.

3.2.2 Iteration 2: Decision-parallel blocks

To better utilize GPU resources, we redesigned our algorithm to maintain a stack of partial assignments, and simplify those partial assignments in parallel CUDA blocks.

```
DPLL_host(F):
    while any assignments:
        pop up to MAX_INSTANCES partial assignments
        copy partial assignments to device
```

```
        launch DPLL_device()
        copy result array from device

        if any SAT instances, return SAT

        for any undecided instances, make a true/false decision
            and push new assignments to stack

    return UNSAT

DPLL_device():

    // unit propagation
    while true:
        if clause unit, update assignment
        break if no clauses were unit

    if any false clauses, anyClausesFalse ← true
    if not satisfied, allClausesSatisfied ← false

    write results to global memory
```

This approach introduces an important hyperparameter, `MAX_INSTANCES`, which controls how many proposed partial assignments can be processed in parallel CUDA blocks. At low values, the algorithm will underutilize the GPU but follow a computation path more similar to sequential DPLL, while higher values will exploit more parallelism, potentially at the cost of lower work efficiency. To start, we set `MAX_INSTANCES = 64`, just high enough to fully occupy the GPUs 46 multiprocessors, as this was our guess at the inflection point where further increase of the parameter would not lead to extra parallelism. We performed a sensitivity study on this parameter for our final implementation in GPU Sensitivity to `MAX_INSTANCES`.

Profiling painted a mixed picture of our performance. The following metrics were obtained from profiling one kernel launch from test case `uuf150-01`:

Compute Throughput	9.51%
Memory Throughput	18.68%
Achieved Occupancy	65%
Avg. Active Threads / Warp	12.90
L1 Hit Rate	40.35%
DRAM Throughput	2.22%
Eligible Warp Rate	20%
Excessive Global Sectors	76%

We observed increased compute, memory throughput, and SM occupancy, but several areas for improvement. Our average active threads per warp was much lower than the optimal 32 at 12.90, indicating significant execution incoherence. Our L1 hit rate was low as well,

signaling that latency may be affecting our speed. At the very least, if there is no way to increase locality, it will probably be worth gathering as much data into shared memory as possible at the beginning of the kernel to hide this latency. The low percentage of eligible warps during scheduling, when compared with the much higher warp occupancy, suggests that warps are spending a significant amount of time stalled, further reinforcing the our suspicion that latency may be bottlenecking our performance. The large number of excessive global sectors indicates that we are not effectively coalescing accesses to global memory between threads.

Digging deeper into instruction-level metrics, we discovered three loads accounting for 45.9% of profiling samples, all of which read literal values from the global assignment. NSight Compute indicated that these stalled samples were almost entirely waiting on global memory loads, indicating that each clauses assignment values were not being cached effectively. We suspect that this is due to the fact that in-between unit literal steps, multiple memory barriers are implied by `syncthread` calls, which evict any cache lines that have any edited literals (CUDA GPUs do not guarantee any form of cache coherency: instead, ensuring visibility with a barrier simply evicts any modified lines from cache). CUDA Compute Capability 7.5 defines a cache line size of 128 bytes, which is in fact almost as large as our assignment array at 150 bytes! One solution would be to pad the assignment array to prevent the entire assignment from being evicted every step, but we opted to just move the current assignment into shared memory at the beginning of the kernel, along with a copy of the formula.

Additionally, we discovered that block-level synchronization after unit propagation was responsible for 11.2% of profiling samples. Increasing the number of blocks by increasing `MAX_INSTANCES` could hide this stall by providing each SM with work to perform during synchronization. However, increasing the number of resident blocks per SM beyond 1 is not possible for formulas with more than 512 clauses, since each SM can only support at most 1024 threads. We decided to keep `MAX_INSTANCES` at 64 for the time being, and explore its effect later with a sensitivity study on our final implementation.

The biggest issue with this iteration, however, was the proportion of time spent copying data between the host and the device. A quick slate of timing experiments revealed that several factors more time was spent copying than in kernels:

Test Case	Kernel Time (ms)	Host Time (ms)
uuf75-01	17.1	2.3
uuf100-01	32.5	5.1
uuf125-01	388.4	65.9

Here, total host time minus kernel time is a reasonable proxy for copy time since copying is the only significant operation done on the host. Reducing the time spent copying would certainly be beneficial. Unfortunately, as long as the host is making literal decisions, copying is inevitable. However, copies can still be interleaved, hiding much of their latency, which is what we aimed to do in the next iteration.

3.2.3 Iteration 3: Optimized decision-parallel blocks

Our final iteration keeps the same structure as iteration 2, but reduces memory access latency and memcpy latency in several ways:

1. We keep the formula, the assignment, and block-level results (`anyPropagated`, `allClausesSatsfied`, etc) in shared memory.
2. We redesigned the formula data structure to store clauses contiguously.
3. We reoriented host side computation to support asynchronous host-device copies.

Keeping copies of global structures in shared memory reduces the latency of repeated access, and storing the formula contiguously allows memory accesses to be coalesced between threads when loading these structures. Profiling reveals that successfully improved many of the metrics we identified in iteration 2:

Compute Throughput	22.94%
Memory Throughput	15.74%
Achieved Occupancy	65%
Avg. Active Threads / Warp	13.04
L1 Hit Rate	40.99%
DRAM Throughput	0.38%
Eligible Warp Rate	39%
Excessive Global Sectors	9%

Our compute throughput improved by roughly a factor of 2, along with the eligible warp rate, suggesting we warps are spending less time stalled, and indicating that we've made progress in removing or hiding latency. While our L1 hit rate remains largely unchanged, our number of accesses to global memory has decreased enormously, from a throughput of 2.22% to 0.38%. We've also reduced memory accesses through coalescing effectively, reducing the number of excessive sectors from 76% to 9%.

Running the same host-device copy experiments as in iteration 2 reveals we have made progress in hiding copy latency through asynchronous interleaving:

Test Case	Kernel Time (ms)	Host Time (ms)
uuf75-01	17.1	2.3
uuf100-01	27.2	4.9
uuf125-01	113.5	26.1

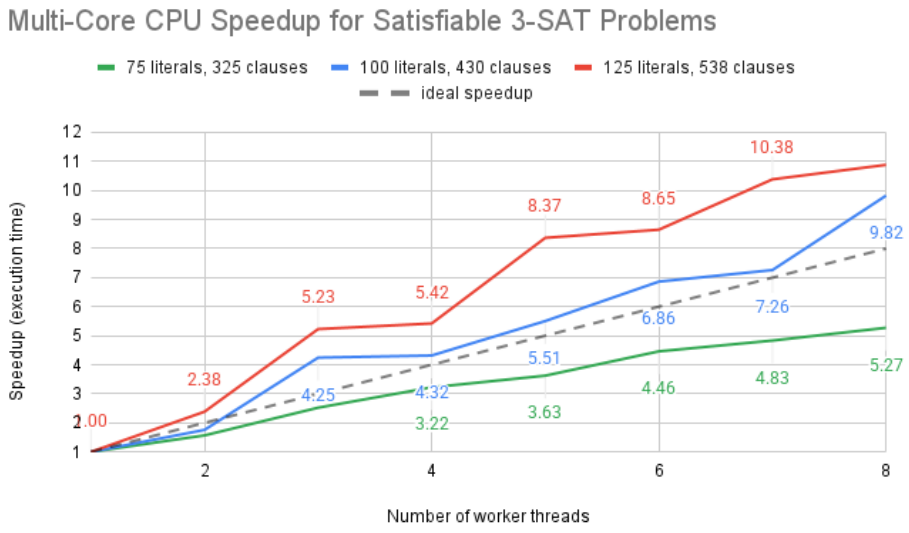
4 Results

All data was gathered on 3-SAT problems from the SATLIB benchmark suite [4]. The benchmark suite has both satisfiable and unsatisfiable problems, which are sorted into buckets based on number of literals and number of clauses. `uf` indicates a satisfiable problem, `uuf` indicates an unsatisfiable problem.

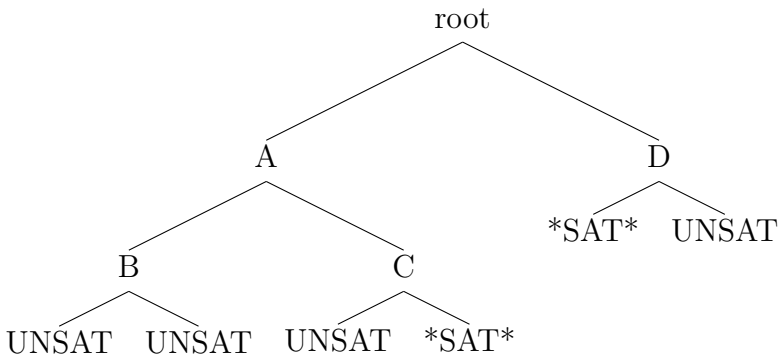
- `uf50-218`, `uuf50-218` (50 literals, 218 clauses)
- `uf75-325`, `uuf75-325` (75 literals, 325 clauses)
- `uf100-430`, `uuf100-430` (100 literals, 430 clauses)
- `uf125-538`, `uuf125-538` (125 literals, 538 clauses)

Within SATLIB buckets, there is large variance in the execution time between various benchmarks. Furthermore, the order in which threads finish their tasks and push to the thread pool stack impacts execution time. To control for these sources of variance, the following data was gathered by averaging the execution time over 10 different benchmarks, and each benchmark was run 3 times.

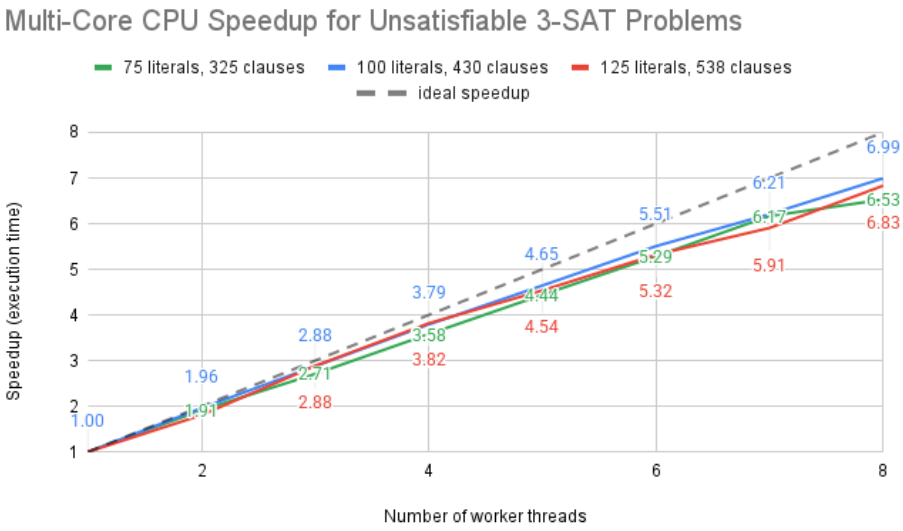
4.1 CPU Speedup



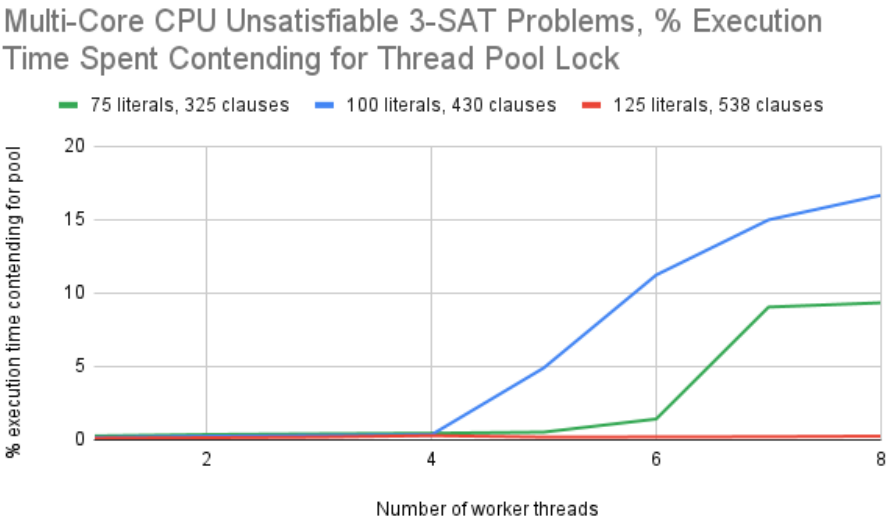
For larger problem sizes (`uf100-430`, `uf125-538`), we achieve super-linear speedup. For an assignment tree, there exist multiple `SAT` nodes at various depths. A single-threaded DFS implementation will always return the first `SAT` node found at some depth `n`. When adding more worker threads, we perform DFS on a wider subtree, so it is possible we will get lucky and find a shallower solution at some depth `m < n`. Consider the following example:



With a small number of worker threads, subtree `A` will be fully explored before moving on to subtree `D`, so we will never find the shallower solution. When adding more worker threads, the exploration of subtree `A` is saturated, so a worker will start exploring subtree `D` and will find the shallower solution. We take advantage of this behavior by having any worker that finds a `SAT` node signal the master thread. This is implemented by having the master thread block on a conditional variable, `satCv`, until any worker thread finds a `SAT` node and notifies the main thread.



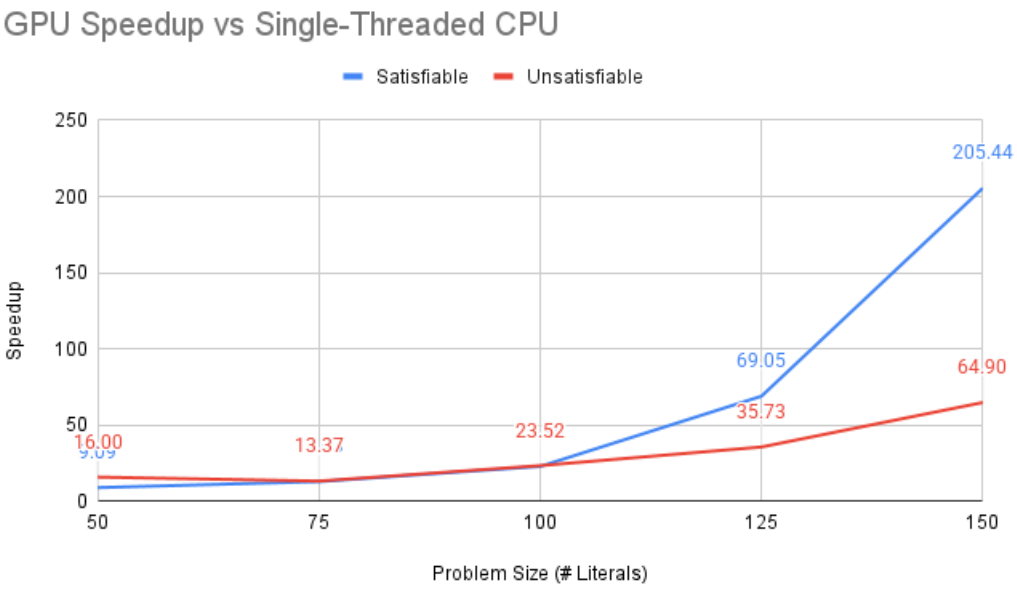
Unsurprisingly, for unsatisfiable problems we achieve nearly linear speedup. Unsatisfiable problems require the entire assignment space to be searched before concluding `UNSAT`, so this is trivially sped up by having workers search different parts of the space. That being said, there is some delta between ideal and observed speedup, which we attribute to contention for the thread pool lock.



The figure above shows the percentage of execution time spent contending for the thread pool lock, per worker. For `uuf75-325` and `uuf100-430`, the time spent contending for the lock is proportional to the gap between ideal and observed speedup. The thread pool is implemented with a single lock, which workers must acquire to: pop a task from the stack, push a task to the task, or signal they completed their task. As the number of worker threads increase, they spend more time waiting to acquire the lock while someone else is holding it, explaining the sub-linear speedup for unsatisfiable problems.

For `uuf125-538`, the time spent contending for the lock is negligible compared to the overall execution time, raising questions about what is causing sub-linear speedup for this benchmark. Further analysis is needed to understand this behavior.

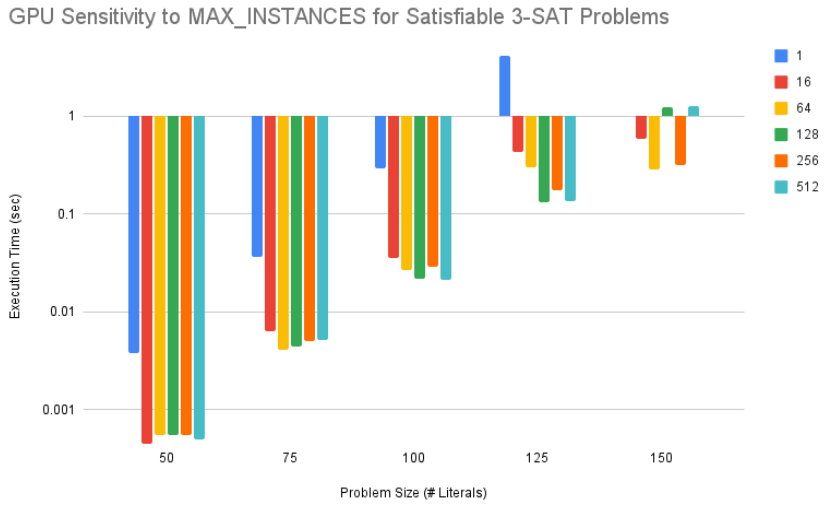
4.2 GPU Speedup

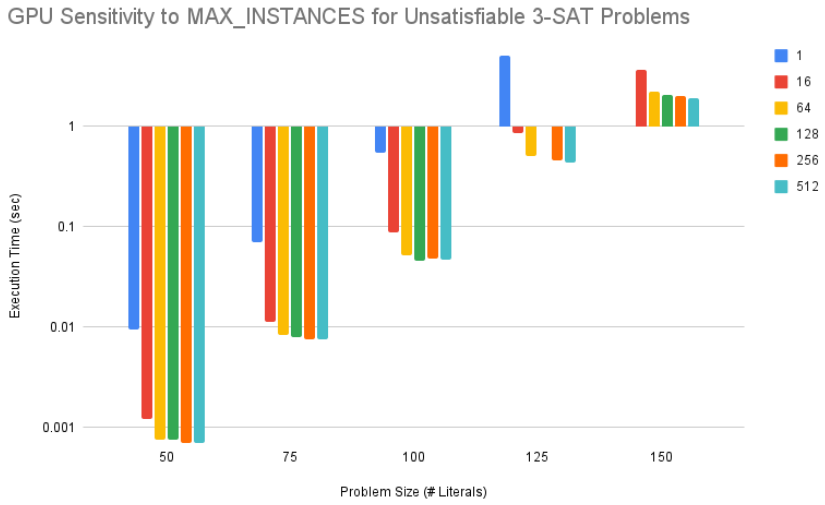


We were able to obtain a speedup of on an NVIDIA RTX 2080 significantly above what we were able to achieve on our multi-core CPU. While this is not an apples-to-apples comparison (GPUs draw more power, for instance), we find the scaling behavior of the speedup encouraging: as the problem size grows, we observe relatively better utilization of the GPUs resources when compared to the multi-core CPU. Additionally, we observe a continuation of the pattern that satisfiable instances produce higher speedups. This aligns with our original expectation that a GPU would more effectively exploit the data-parallel inner loops of DPLL, which grow with problem size.

One way we can support this claim is to look at kernel durations. We see highly sublinear increases in kernel durations as problem size increases; at a problem size of 50 literals (218 clauses), typical kernel durations are around 25μs, while at a problem size of 150 literals (645 clauses), kernel durations are still only around 40μs. This indicates that our kernel is effectively exploiting the data-parallelism within each step, something our multi-core CPU implementation does not. At 8 threads, there is only a 2x difference between the multi-core speedup on a 50 literal satisfiable problem and a 125 literal satisfiable problem (10.38x vs 5.27x), compared to a difference of 23x on the GPU (205.44 vs 9.09x).

4.3 GPU Sensitivity to MAX_INSTANCES





To visualize differences at a variety of time scales, both graphs have been displayed with a log time axis. Additionally, timing tests for ***MAX_INSTANCES****=1 and 150 literals did not terminate and data was not collected.*

Our initial guess of 64 for `MAX_INSTANCES` appears to have been reasonable. After 64, performance plateaus, particularly for satisfiable instances. Interestingly, profiling reveals this may not be for the reason we predicted (i.e., that GPU utilization plateaus as well). Compute throughput increases from 21% at `MAX_INSTANCES`=64 to 35% at `MAX_INSTANCES`=512. However, the amount of data copied between the host and the device increases as the number of instances increases. We can probe this indirectly by looking at the ratio of kernel time to host time (gathered on test case `uuf125-01`):

MAX_INSTANCES	Kernel Time (ms)	Host Time (ms)
64	25.5	96.8
128	24.6	109.0
256	20.1	117.7
512	17.0	112.3

Even though kernel time decreases as `MAX_INSTANCES` increases, the time on the host (almost all of which is copying) increases as the result array grows and the number of assignments grows, all of which must be copied on before a kernel launch.

References

[1] Wikipedia contributors. *Boolean satisfiability problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-April-2025]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1286671443.

[2] Wikipedia contributors. *DPLL algorithm — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=DPLL_algorithm&oldid=1276987229. [Online; accessed 28-April-2025]. 2025.

[3] Matt Fredrikson and Andre Platzer. *Lecture Notes on SAT Solvers & DPLL*. URL: https://www.cs.cmu.edu/~15414/f17/lectures/10-dpll.pdf.

[4] Holger H. Hoos and Thomas Stützle. *SATLIB: Benchmark Problems for SAT*. https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html. [Online; accessed 28-April-2025].