Description of the implementation

Algorithms: Spatial Partitioning and Collision Resolution

For each simulation step, collision detection and resolution occur in two main stages:

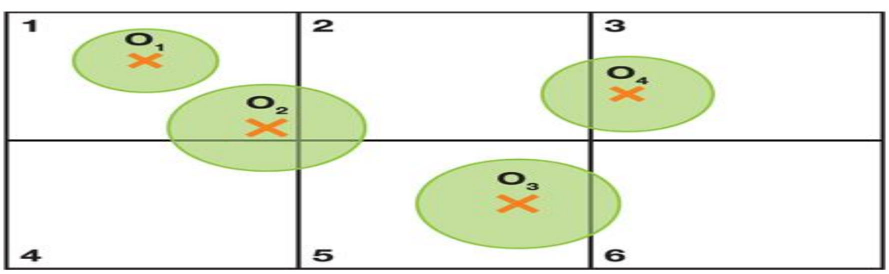
1. Wall Collisions: Particles near the simulation boundaries are checked and resolved by reflecting their velocity components.
2. Particle-Particle Collisions: Overlapping particles are detected and resolved using elastic collision equations to ensure conservation of momentum and energy.

This process **repeats iteratively** until no further collisions remain in the current timestep.

The simulator employs a **spatial partitioning technique**, specifically a **broad-phase collision detection** approach, to efficiently identify potential collisions. Instead of checking all possible particle pairs (which are computationally expensive, requiring O(N^2) complexity), the simulation divides the simulation area into a uniform grid. Each cell in this grid contains only particles within its boundaries, significantly reducing the number of comparisons.

Each particle is assigned to a grid cell based on its position. To detect potential collisions, the algorithm considers only particles located in the Moore Neighborhood (8-neighborhood) of a given grid cell. This includes: The current cell (self-check). The 8 adjacent cells: top-left, top, top-right, left, right, bottom-left, bottom, bottom-right. However, instead of checking all possible pairs within the Moore Neighborhood, the algorithm first builds a candidate list containing only particle pairs that overlap. After this pre-processing, the simulation then checks and resolves only the collisions present in the candidate list.

Here is an example: (*Image by Nvidia-Developer)*



The pairs we need to check are only (O1, O2), (O2, O3), (O3, O4).

Data structures:

1. **Grid-Based Spatial Partitioning (std::vector<std::vector<Particle*>> grid).** Each element in this vector is the list of particles that are in these cells.
2. **Candidate Collision List (std::vector<std::pair<Particle*, Particle*>> candidate[4].** Stores only overlapping particle pairs before checking for collisions.

Parallelism strategy:

+) OpenMP Constructs Used

- #pragma omp parallel for collapse(2):
  Used in functions like resolve_wall_collisions and when generating candidate collisions in build_possible_collisions. This helps achieve better load balancing among threads because the entire grid is divided evenly across available threads.
- #pragma omp parallel for:
  Applied in resolve_adjacent_collisions to iterate over the candidate collision list. This construct allows concurrent processing of multiple pairs without data conflicts.
- #pragma omp critical:
  Employed inside build_possible_collisions when adding a candidate pair to the shared vector. Because multiple threads might find overlapping candidate pairs concurrently, the critical section ensures that the insertion into the shared candidate list is done safely to avoid data races.

+) How Work Is Divided Among Threads

- The simulation area is divided into a uniform grid (flattened into a 1D vector for better memory locality). Each grid cell holds pointers to the particles it contains.
- Interleaved case: The grid is further partitioned into four interleaved cases (based on even/odd row and column indices). This division ensures that each thread works on a disjoint set of grid cells, preventing overlapping access and thus data races.
- Candidate List Generation and Processing: Each of the four interleaved cases produces its own candidate collision list. Later, collision resolution on these lists (in resolve_adjacent_collisions) is performed independently and in parallel.

+) Synchronization Handling

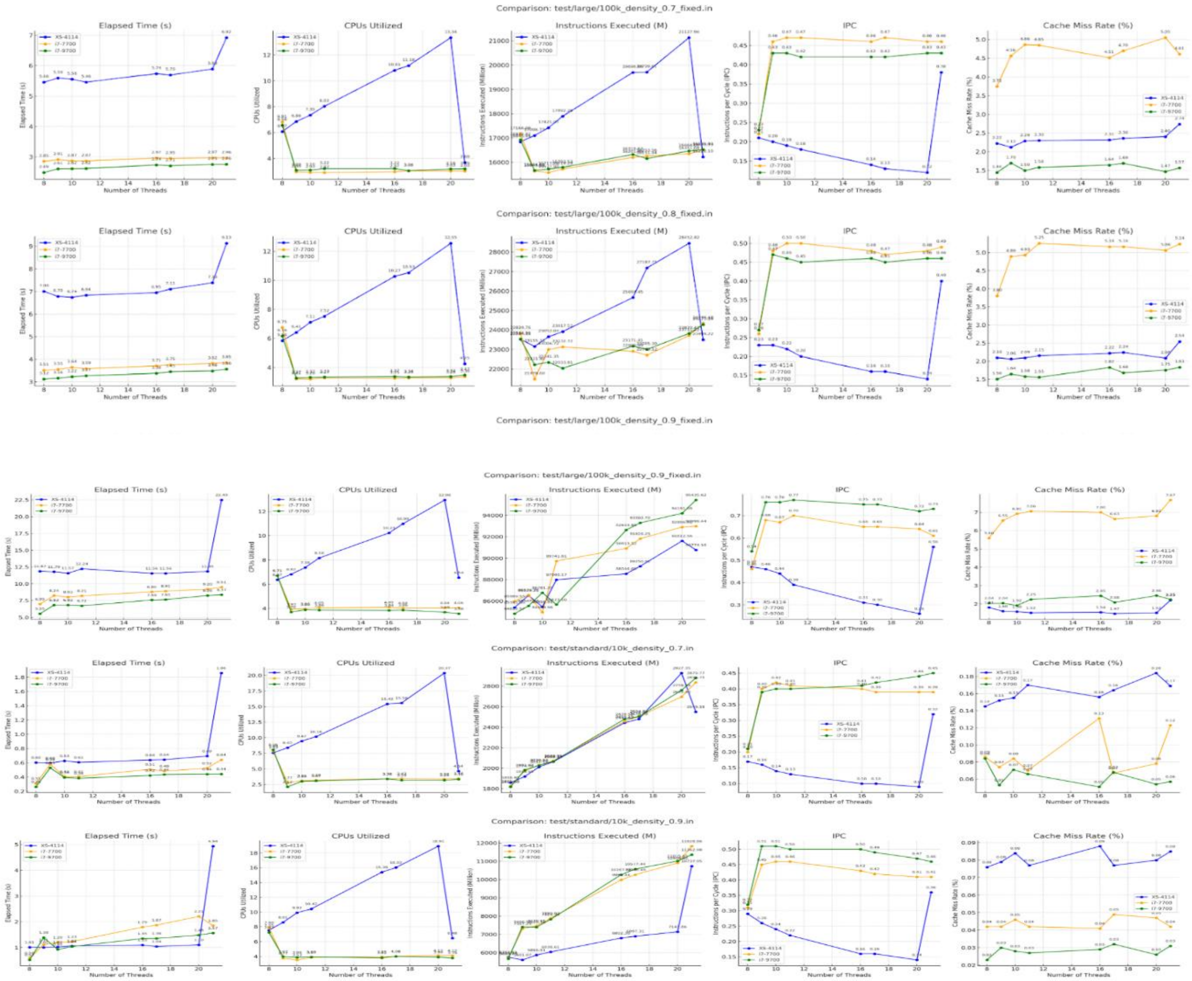
- Critical Sections: In build_possible_collisions, candidate pairs are added to the shared vector using a critical section (#pragma omp critical) to protect against concurrent insertions. This minimizes race conditions during candidate list formation. By dividing the grid into four interleaved cases, each thread works on an independent candidate list. This design minimizes the need for synchronization during the collision resolution phase, as no two threads update the same candidate list concurrently.

Description, visualization, and data on your execution

Merge graph of 3 hardware type (xs-4114, i7-7700, i7-9700) regard to their performance and their key index’s performance. (Using graph from appendix 1)

+) **Effect of Input Parameters on Performance:**

Because our program is dependent on OpenMP for multithreading execution, which can boost the rate of task processing with the use of more than a single CPU core => Thread number (params.param_threads) has a great influence on running time.



Number of Threads: It is improved to 16 threads. For more than 16, performance neither increases nor improves due to the overhead of synchronization and memory saturation. The benefit of adding threads starts diminishing at 16 threads because Instructions per Cycle (IPC) drops and cache miss rates increase, thereby enhancing memory access inefficiency. When 21 threads are run, the increased overhead of synchronizing and uncalled-for access to memory results in the slow-down of the program instead of speeding it up.

Test Density & Input File Type Effect: High test density inputs (0.9 vs. 0.7) have the effect of lowering IPC, i.e., computations are of higher complexity and longer times are taken for their execution. Greater test inputs (100k vs. 10k) also have a considerably larger effect on processing time through more computations and memory accesses and cause more cache misses.

Cache Miss Rate Impacts: Increasing the number of threads causes more cache misses, peaking to 2.7% in bigger tests. This indicates that the program is approaching memory bandwidth constraints when multiple threads attempt to access data simultaneously.

Code Effect on Performance: The program relies heavily on OpenMP for parallel computation, particularly resolve_wall_collisions() resolve_adjacent_collisions() build_possible_collisions().Using #pragma omp parallel for allows several threads to process particles concurrently, reducing time taken to detect collisions and utilizing CPU cores more effectively. But when more than excessive threads are used, memory contention is greater, which limits further performance improvements above 16 threads. The #pragma omp critical block in build_possible_collisions() keeps multiple threads from modifying common collision data at the same time, but also creates contention, reducing performance benefits at higher numbers of threads. Performance graphs confirm that increasing the number of threads to 16 increases the speed of execution but using more than this number of threads makes performance sluggish due to the limitations of memory and synchronization.

+) **Different hardware performance Comparison**

The xs-4114 generally has slower execution times compared to i7 processors, especially for higher thread counts. It has higher CPU utilization, meaning it uses more available cores. While it performs well in large tests, it struggles in smaller, standard tests. The i7-7700 (Consumer-Grade 7th Gen CPU) has a moderate execution time, often better than the xs-4114 but not as good as the i7-9700. It has lower CPU utilization (fewer cores), but its higher IPC compensates. However, it struggles with cache efficiency, leading to higher cache miss rates. The i7-9700 (Consumer-Grade 9th Gen CPU) delivers the best performance overall in execution time. It has more efficient CPU utilization due to improved architecture, higher IPC, and lower cache miss rates, making it the most efficient among the three.

IPC measures how efficiently the CPU executes instructions. The i7-9700 has the highest IPC (~0.76 in large tests), making it the most efficient. The i7-7700 follows closely, but its higher cache miss rate reduces efficiency. The xs-4114 has the lowest IPC (~0.2-0.4), meaning it takes more cycles per instruction, contributing to slower execution. A high cache miss rate results in more memory access, slowing down execution. The xs-4114 has a moderate cache miss rate (1.5% - 2.4%), the i7-7700 has a higher cache miss rate (4%-7%), which negatively impacts performance, and the i7-9700 has the lowest cache miss rate (~2%), benefiting from better memory locality.

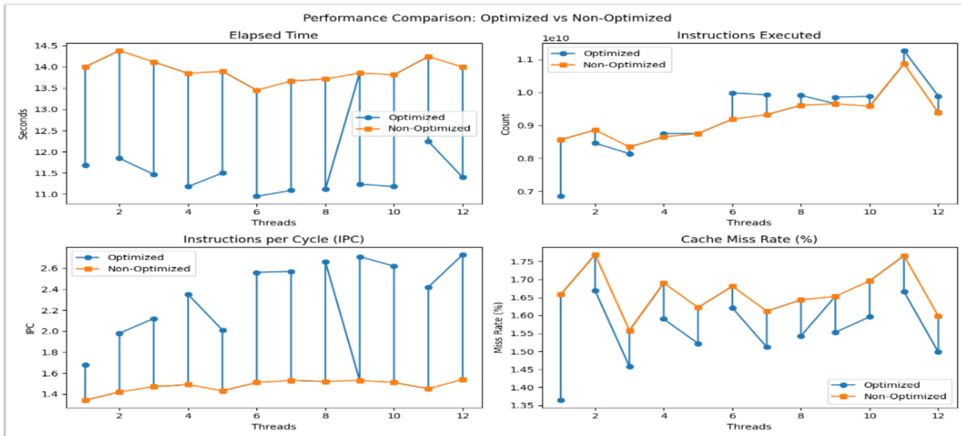
The task clock indicates the total CPU time spent on computation. The xs-4114 has high task clock usage, meaning it takes longer to complete a task. The i7-7700 has medium task clock usage but is still affected by cache inefficiency. The i7-9700 has the lowest task clock usage, indicating the most efficient computation per cycle. The performance differences observed are influenced by key hardware characteristics. The xs-4114 has more cores but a lower clock speed and lower IPC, making it slower despite using many threads. The i7-7700 has a higher clock speed but fewer cores, meaning it performs better than the xs-4114 in single-threaded tasks. The i7-9700 combines high clock speed, efficient IPC, and low cache misses, making it the best-performing CPU overall. The code relies on OpenMP for parallel execution. A higher core count (as seen in the xs-4114) should be an advantage, but low IPC and high memory latency reduce the benefit. The i7-9700 balances core count and IPC, making it the most efficient processor for this workload.

Two performance optimizations

1. Removed pragma atomic in

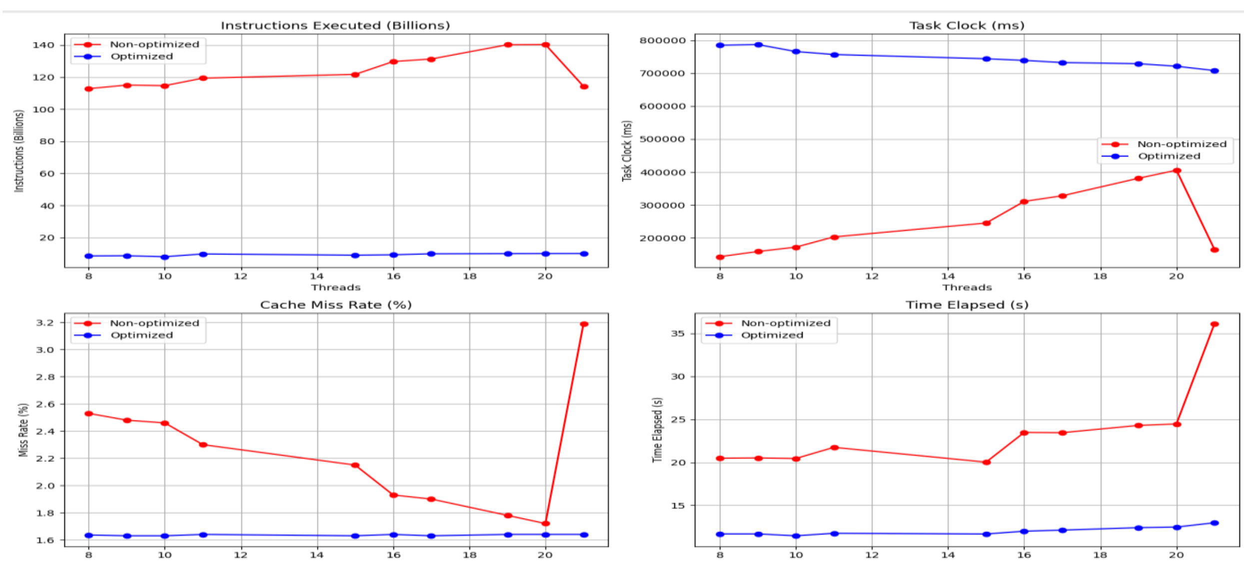
```
41      -    #pragma omp atomic
42      -    collisionOccurred |= true;
    41  +    collisionOccurred = true;
```

Reasoning Behind the Optimization: The OpenMP atomic directive was originally used to protect collisionOccurred from race conditions. However, removing #pragma omp atomic and directly assigning collisionOccurred = true; did not introduce correctness issues because: Race conditions do not impact correctness in this case – If multiple threads set collisionOccurred = true; simultaneously, the result remains the same: the flag will be set. Result: By removing it, the program achieves higher parallel efficiency. Instructions executed were reduced, as synchronization-related overhead was removed. IPC improved, indicating better instruction efficiency per cycle. Cache miss rates remained similar, suggesting the optimization did not introduce memory contention issues. (Graph from Appendix 6)



2. Checking wall collisions for particles that belong to grids that are near the walls

We restrict our wall collision checks to particles in grid cells along the four edges of the simulation area (i.e., cells where row_id == 0, column_id == 0, row_id == grid_size - 1, or column_id == grid_size - 1). This approach minimizes unnecessary computation by avoiding checks on interior cells, where wall collisions cannot occur. The optimization shows a clear reduction in elapsed time, fewer instructions executed, and a higher IPC, all indicating more efficient CPU usage. Notably, cache miss rates also have slightly improved, suggesting no memory bottlenecks. Here is the graph from raw data in Appendix 7.

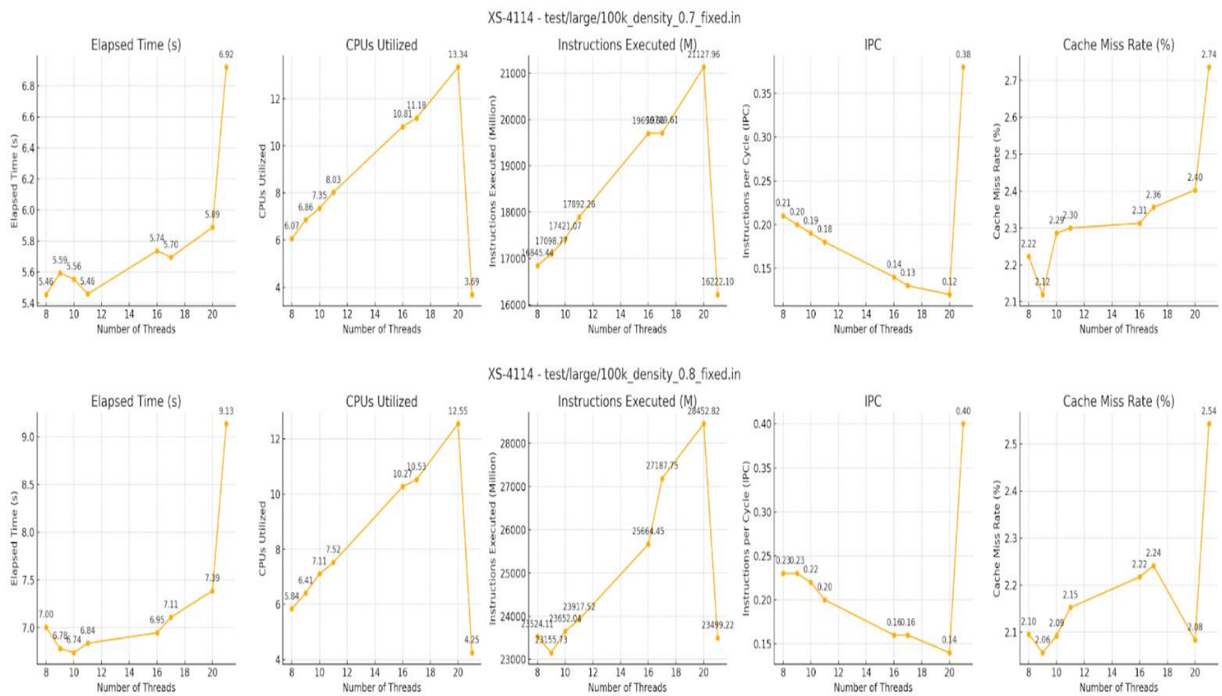


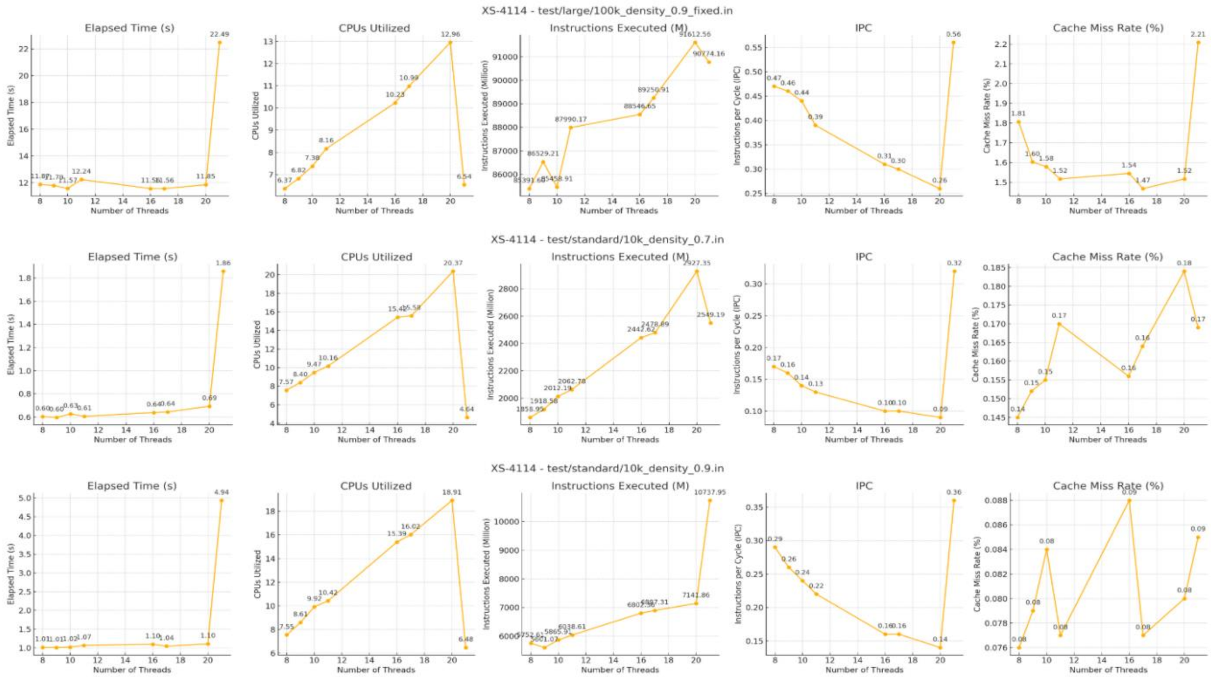
Appendix

The reports and codes are done by Le Huy Chau, A0276221L and Tran Khoi Nguyen, A0276180A.

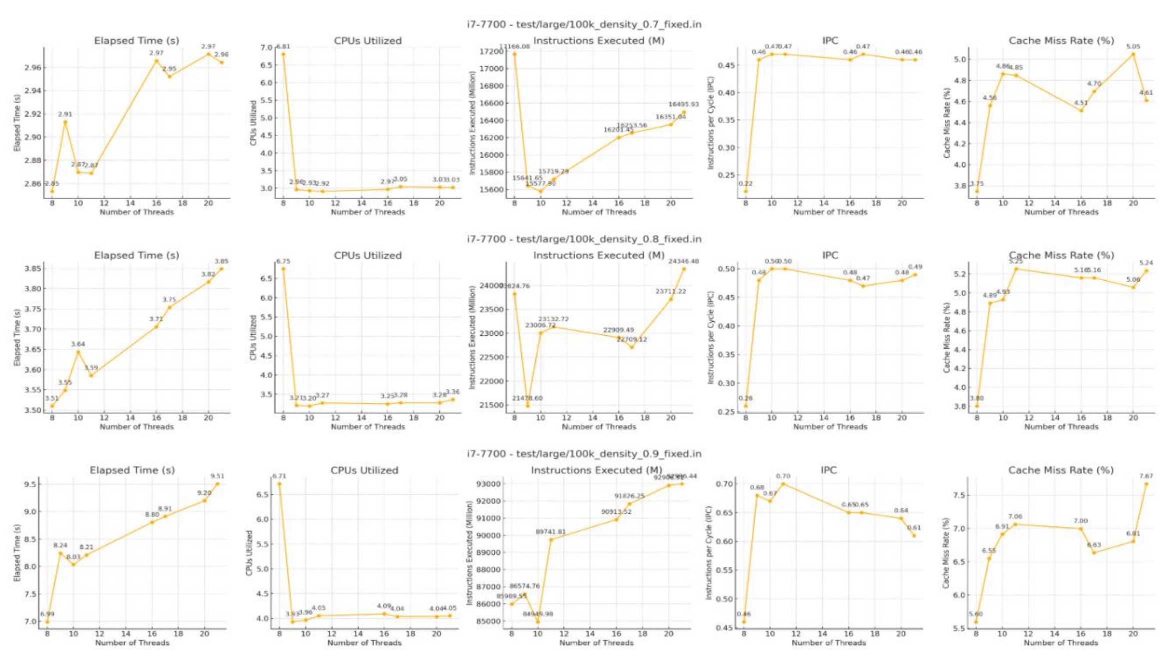
Algorithms are implemented based on this reading:
https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-32-broad-phase-collision-detection-cuda

Appendix 1: XS-4114, i7-7700, i7-9700 Graph about its performance among 5 test cases, repeat 5 times, for each thread used:”(8 9 10 11 16 17 20 21)”, from table data of Appendix 2,3,4

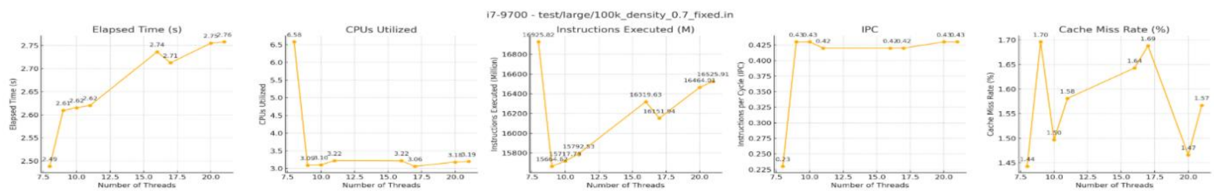


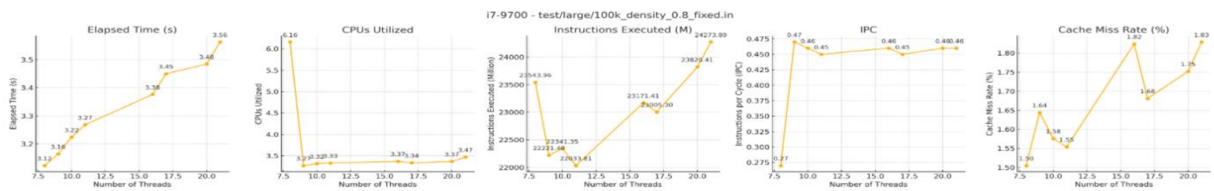


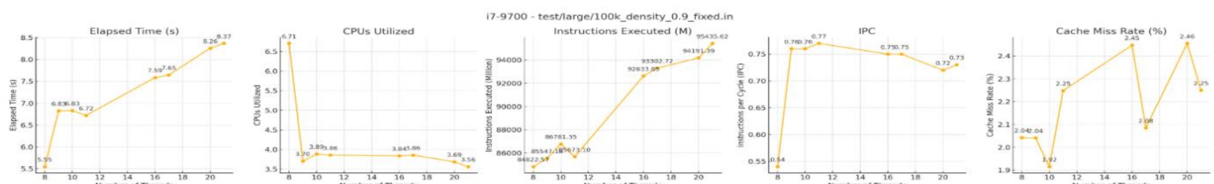
i7-7700 Graph about its performance among several test cases

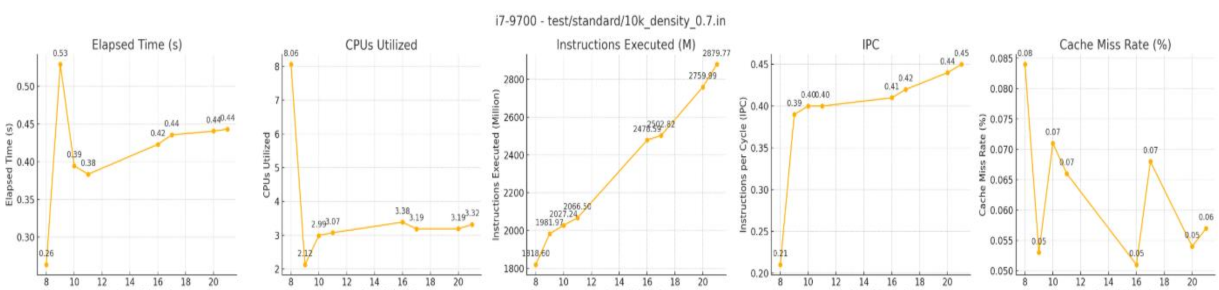


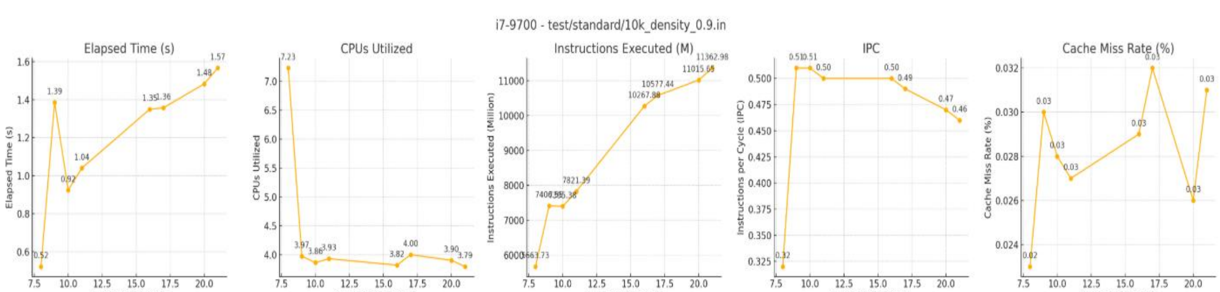
i7-9700 Graph about its performance among several test cases











Appendix 2

Run (repeat 5 times) xs-4114 for 5 tests in large and standard with thread used : (8 9 10 11 16 17 20 21)

slurm_job code:

```
#!/bin/bash

## This is an example Slurm template job script for A1 that just runs the script and arguments you pass in via `srun`.

#SBATCH --job-name=test
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --mem=4gb
#SBATCH --partition=xs-4114
#SBATCH --time=00:10:00
#SBATCH --output=%x_%j.slurmlog
#SBATCH --error=%x_%j.slurmlog

echo "We are running on $(hostname)"
echo "Job started at $(date)"
```

```
# Define test cases
declare -a tests=("tests/large/100k_density_0.7_fixed.in"
                  "tests/large/100k_density_0.8_fixed.in"
                  "tests/large/100k_density_0.9_fixed.in"
                  "tests/standard/10k_density_0.7.in"
                  "tests/standard/10k_density_0.9.in")

# Define thread counts
declare -a threads=(8 9 10 11 15 16 17 19 20 21)

make

# Run perf stat for each test case with each thread count
for test in "${tests[@]}"; do
    for t in "${threads[@]}"; do
        echo "Running perf stat on $test with $t threads"
        perf stat --repeat 5 -- ./sim.perf "$test" "$t"
    done
done

echo "Job ended at $(date)"
```

Elapsed Time (seconds)	CPUs Utilized	Instructions Executed	Instructions per Cycle (IPC)	Cache Miss Rate (%)	Number of Threads Used	Type of Test Used	Task Clock (seconds)
5.456	6.072	16845436367	0.21	2.223	8	tests/large/100k_density_0.7_fixed.in	33.12921
5.5938	6.855	17098766792	0.2	2.12	9	tests/large/100k_density_0.7_fixed.in	38.34629
5.5555	7.35	17421072895	0.19	2.286	10	tests/large/100k_density_0.7_fixed.in	40.83518
5.4594	8.027	17892257087	0.18	2.3	11	tests/large/100k_density_0.7_fixed.in	43.82307
5.7368	10.814	19696664051	0.14	2.313	16	tests/large/100k_density_0.7_fixed.in	62.03574
5.6954	11.177	19709611836	0.13	2.356	17	tests/large/100k_density_0.7_fixed.in	63.65453
5.8879	13.336	21127963046	0.12	2.403	20	tests/large/100k_density_0.7_fixed.in	78.52206
6.9194	3.695	16222098931	0.38	2.736	21	tests/large/100k_density_0.7_fixed.in	25.56696
7.0047	5.839	23524108776	0.23	2.095	8	tests/large/100k_density_0.8_fixed.in	40.89823
6.784	6.41	23155734614	0.23	2.056	9	tests/large/100k_density_0.8_fixed.in	43.48017
6.74	7.108	23652041706	0.22	2.092	10	tests/large/100k_density_0.8_fixed.in	47.90494
6.839	7.521	23917521761	0.2	2.152	11	tests/large/100k_density_0.8_fixed.in	51.4344
6.9475	10.271	25664454939	0.16	2.217	16	tests/large/100k_density_0.8_fixed.in	71.36025
7.11	10.526	27187746098	0.16	2.241	17	tests/large/100k_density_0.8_fixed.in	74.84279
7.3859	12.55	28452820747	0.14	2.083	20	tests/large/100k_density_0.8_fixed.in	92.68893
9.134	4.251	23499224353	0.4	2.543	21	tests/large/100k_density_0.8_fixed.in	38.82807
11.872	6.367	85391597072	0.47	1.807	8	tests/large/100k_density_0.9_fixed.in	75.58896
11.792	6.818	86529205820	0.46	1.603	9	tests/large/100k_density_0.9_fixed.in	80.39935
11.5682	7.375	85458906672	0.44	1.579	10	tests/large/100k_density_0.9_fixed.in	85.31143
12.238	8.162	87990171280	0.39	1.517	11	tests/large/100k_density_0.9_fixed.in	99.88184
11.559	10.232	88546652946	0.31	1.545	16	tests/large/100k_density_0.9_fixed.in	118.26583
11.559	10.987	89250910276	0.3	1.468	17	tests/large/100k_density_0.9_fixed.in	127.00102
11.848	12.956	91612563002	0.26	1.517	20	tests/large/100k_density_0.9_fixed.in	153.51134
22.4855	6.54	90774162034	0.56	2.209	21	tests/large/100k_density_0.9_fixed.in	147.06613
0.6038	7.568	1858951352	0.17	0.145	8	tests/standard/10k_density_0.7.in	4.56948
0.5969	8.399	1918579868	0.16	0.152	9	tests/standard/10k_density_0.7.in	5.01329
0.6277	9.469	2012189662	0.14	0.155	10	tests/standard/10k_density_0.7.in	5.94324
0.6068	10.165	2062781440	0.13	0.17	11	tests/standard/10k_density_0.7.in	6.16779
0.639	15.417	2442621168	0.1	0.156	16	tests/standard/10k_density_0.7.in	9.8514
0.6441	15.579	2478887244	0.1	0.164	17	tests/standard/10k_density_0.7.in	10.03495
0.6939	20.369	2927348344	0.09	0.184	20	tests/standard/10k_density_0.7.in	14.13454
1.8579	4.639	2549190607	0.32	0.169	21	tests/standard/10k_density_0.7.in	8.61949
1.0132	7.548	5752606758	0.29	0.076	8	tests/standard/10k_density_0.9.in	7.64742
1.0071	8.606	5601067659	0.26	0.079	9	tests/standard/10k_density_0.9.in	8.6671
1.0201	9.918	5865908899	0.24	0.084	10	tests/standard/10k_density_0.9.in	10.11753
1.0674	10.424	6038612595	0.22	0.077	11	tests/standard/10k_density_0.9.in	11.12719
1.0964	15.394	6802357918	0.16	0.088	16	tests/standard/10k_density_0.9.in	16.87827
1.0448	16.019	6897307330	0.16	0.077	17	tests/standard/10k_density_0.9.in	16.73682
1.105	18.91	7141857142	0.14	0.08	20	tests/standard/10k_density_0.9.in	20.89471
4.9355	6.478	10737951200	0.36	0.085	21	tests/standard/10k_density_0.9.in	31.9707

Appendix 3: i7-7700

#!/bin/bash

This is an example Slurm template job script for A1 that just runs the script and arguments you pass in via `srun`.

#SBATCH --job-name=test

#SBATCH --nodes=1

#SBATCH --ntasks=1

#SBATCH --mem=4gb

```
#SBATCH --partition=i7-7700

#SBATCH --time=00:30:00

#SBATCH --output=%x_%j.slurmlog

#SBATCH --error=%x_%j.slurmlog


echo "We are running on $(hostname)"

echo "Job started at $(date)"


# Define test cases

declare -a tests=("tests/large/100k_density_0.7_fixed.in"

                  "tests/large/100k_density_0.8_fixed.in"

                  "tests/large/100k_density_0.9_fixed.in"

                  "tests/standard/10k_density_0.7.in"

                  "tests/standard/10k_density_0.9.in")


# Define thread counts

declare -a threads=(8 9 10 11 16 17 20 21)


make


# Run perf stat for each test case with each thread count

for test in "${tests[@]}"; do

    for t in "${threads[@]}"; do

        echo "Running perf stat on $test with $t threads"

        srun --partition=i7-7700 perf stat -r 5 -e task-clock,cycles,instructions,cache-misses,cache-
references -- ./sim.perf "$test" "$t"

    done

done


echo "Job ended at $(date)"
```

Elapsed Time (seconds)	CPUs Utilized	Instructions Executed	Instructions per Cycle (IPC)	Cache Miss Rate (%)	Number of Threads Used	Type of Test Used	Task Clock (seconds)
2.8533	6.808	17166075427	0.22	3.747	8	tests/large/100k_density_0.7_fixed.i	19.42566
2.91299	2.965	15641648714	0.46	4.561	9	tests/large/100k_density_0.7_fixed.i	8.63832
2.86973	2.932	15577903671	0.47	4.863	10	tests/large/100k_density_0.7_fixed.i	8.41468
2.8689	2.915	15719285505	0.47	4.848	11	tests/large/100k_density_0.7_fixed.i	8.36275
2.9658	2.97	16201450587	0.46	4.513	16	tests/large/100k_density_0.7_fixed.i	8.80755
2.9521	3.046	16253560841	0.47	4.696	17	tests/large/100k_density_0.7_fixed.i	8.99147
2.97137	3.03	16351035381	0.46	5.049	20	tests/large/100k_density_0.7_fixed.i	9.00256
2.96442	3.026	16495931464	0.46	4.611	21	tests/large/100k_density_0.7_fixed.i	8.96958
3.51	6.747	23824764077	0.26	3.803	8	tests/large/100k_density_0.8_fixed.i	23.68129
3.5488	3.21	21478600171	0.48	4.892	9	tests/large/100k_density_0.8_fixed.i	11.39103
3.6437	3.197	23006716192	0.5	4.928	10	tests/large/100k_density_0.8_fixed.i	11.65013
3.5851	3.274	23132718151	0.5	5.254	11	tests/large/100k_density_0.8_fixed.i	11.73772
3.70587	3.252	22909489512	0.48	5.159	16	tests/large/100k_density_0.8_fixed.i	12.05216
3.7536	3.28	22709124557	0.47	5.16	17	tests/large/100k_density_0.8_fixed.i	12.31264
3.8165	3.28	23711224882	0.48	5.059	20	tests/large/100k_density_0.8_fixed.i	12.51981
3.8488	3.364	24346476780	0.49	5.235	21	tests/large/100k_density_0.8_fixed.i	12.94586
6.9895	6.714	85989551601	0.46	5.596	8	tests/large/100k_density_0.9_fixed.i	46.92521
8.244	3.929	86574764542	0.68	6.55	9	tests/large/100k_density_0.9_fixed.i	32.38733
8.0333	3.965	84949980481	0.67	6.914	10	tests/large/100k_density_0.9_fixed.i	31.84863
8.21	4.053	89741807449	0.7	7.062	11	tests/large/100k_density_0.9_fixed.i	33.27533
8.803	4.091	90913520471	0.65	6.998	16	tests/large/100k_density_0.9_fixed.i	36.00923
8.9126	4.036	91826250976	0.65	6.635	17	tests/large/100k_density_0.9_fixed.i	35.96928
9.1969	4.039	92906912510	0.64	6.806	20	tests/large/100k_density_0.9_fixed.i	37.14862
9.5058	4.052	92996442094	0.61	7.671	21	tests/large/100k_density_0.9_fixed.i	38.51723
0.30751	7.841	1825487627	0.2	0.086	8	tests/standard/10k_density_0.7.in	2.41126
0.5782	2.772	1974899092	0.4	0.074	9	tests/standard/10k_density_0.7.in	1.60275
0.40675	3.108	2030625039	0.42	0.084	10	tests/standard/10k_density_0.7.in	1.26411
0.40645	3.189	2070709995	0.41	0.07	11	tests/standard/10k_density_0.7.in	1.29619
0.50917	3.393	2455191477	0.4	0.131	16	tests/standard/10k_density_0.7.in	1.72777
0.48326	3.433	2504738749	0.39	0.067	17	tests/standard/10k_density_0.7.in	1.65895
0.52499	3.371	2695395760	0.39	0.078	20	tests/standard/10k_density_0.7.in	1.76994
0.64263	3.461	2836754020	0.39	0.123	21	tests/standard/10k_density_0.7.in	2.22401
0.5923	7.902	5705884945	0.31	0.042	8	tests/standard/10k_density_0.9.in	4.68054
1.1106	3.765	7307251244	0.45	0.042	9	tests/standard/10k_density_0.9.in	4.18168
1.2028	3.529	7470947718	0.46	0.046	10	tests/standard/10k_density_0.9.in	4.24509
1.2289	3.872	7880918739	0.46	0.042	11	tests/standard/10k_density_0.9.in	4.75757
1.7937	3.95	9986330694	0.43	0.041	16	tests/standard/10k_density_0.9.in	7.08514
1.8713	4.012	10268485737	0.42	0.049	17	tests/standard/10k_density_0.9.in	7.50787
2.2096	4.123	10908621513	0.41	0.047	20	tests/standard/10k_density_0.9.in	9.10898
1.85487	4.117	11828858277	0.41	0.042	21	tests/standard/10k_density_0.9.in	7.63714

Appendix 4: i7-9700

slurm job

#!/bin/bash

This is an example Slurm template job script for A1 that just runs the script and arguments you pass in via `srun`.

```
#SBATCH --job-name=test

#SBATCH --nodes=1

#SBATCH --ntasks=1

#SBATCH --mem=4gb

#SBATCH --partition=i7-9700

#SBATCH --time=00:30:00

#SBATCH --output=%x_%j.slurmlog

#SBATCH --error=%x_%j.slurmlog


echo "We are running on $(hostname)"

echo "Job started at $(date)"


# Define test cases

declare -a tests=("tests/large/100k_density_0.7_fixed.in"

                  "tests/large/100k_density_0.8_fixed.in"

                  "tests/large/100k_density_0.9_fixed.in"

                  "tests/standard/10k_density_0.7.in"

                  "tests/standard/10k_density_0.9.in")


# Define thread counts

declare -a threads=(8 9 10 11 16 17 20 21)


make


# Run perf stat for each test case with each thread count

for test in "${tests[@]}"; do

    for t in "${threads[@]}"; do

        echo "Running perf stat on $test with $t threads"

        srun --partition=i7-9700 perf stat -r 5 -e task-clock,cycles,instructions,cache-misses,cache-
references -- ./sim.perf "$test" "$t"
```

```
    done
done

echo "Job ended at $(date)"
```

Elapsed Time (seconds	CPUs Utilized	Instructions Executed	Instructions per Cycle (IP(	Cache Miss Rate (%)	Number o	Type of Test Used	Task Clock (seconds)
2.4883	6.578	16925823375	0.23	1.443	8	tests/large/100k_density_0.7_fixed.in	16.36874
2.6096	3.09	15664618913	0.43	1.696	9	tests/large/100k_density_0.7_fixed.in	8.06422
2.6154	3.103	15717730104	0.43	1.497	10	tests/large/100k_density_0.7_fixed.in	8.11483
2.6201	3.22	15792534054	0.42	1.581	11	tests/large/100k_density_0.7_fixed.in	8.43557
2.7362	3.217	16319630504	0.42	1.643	16	tests/large/100k_density_0.7_fixed.in	8.80372
2.7125	3.061	16151943890	0.42	1.688	17	tests/large/100k_density_0.7_fixed.in	8.30422
2.755	3.183	16464009404	0.43	1.466	20	tests/large/100k_density_0.7_fixed.in	8.76985
2.7578	3.195	16525912563	0.43	1.567	21	tests/large/100k_density_0.7_fixed.in	8.81068
3.1227	6.16	23543962163	0.27	1.505	8	tests/large/100k_density_0.8_fixed.in	19.23733
3.1644	3.272	22221395851	0.47	1.644	9	tests/large/100k_density_0.8_fixed.in	10.35516
3.2242	3.317	22341354883	0.46	1.576	10	tests/large/100k_density_0.8_fixed.in	10.69601
3.2685	3.333	22033809699	0.45	1.554	11	tests/large/100k_density_0.8_fixed.in	10.89545
3.3762	3.37	23171414919	0.46	1.824	16	tests/large/100k_density_0.8_fixed.in	11.37926
3.4502	3.336	23005301718	0.45	1.681	17	tests/large/100k_density_0.8_fixed.in	11.50949
3.4847	3.369	23820405116	0.46	1.752	20	tests/large/100k_density_0.8_fixed.in	11.7403
3.5634	3.47	24273891254	0.46	1.829	21	tests/large/100k_density_0.8_fixed.in	12.36455
5.5479	6.705	84822573631	0.54	2.042	8	tests/large/100k_density_0.9_fixed.in	37.20074
6.8254	3.702	85547176179	0.76	2.041	9	tests/large/100k_density_0.9_fixed.in	25.26485
6.8297	3.886	86781354622	0.76	1.915	10	tests/large/100k_density_0.9_fixed.in	26.53792
6.7211	3.856	85673096806	0.77	2.248	11	tests/large/100k_density_0.9_fixed.in	25.91616
7.586	3.838	92633848067	0.75	2.447	16	tests/large/100k_density_0.9_fixed.in	29.11818
7.647	3.858	93302724407	0.75	2.085	17	tests/large/100k_density_0.9_fixed.in	29.50247
8.257	3.685	94191385729	0.72	2.455	20	tests/large/100k_density_0.9_fixed.in	30.42693
8.3679	3.562	95435617725	0.73	2.25	21	tests/large/100k_density_0.9_fixed.in	29.80923
0.26348	8.059	1818600732	0.21	0.084	8	tests/standard/10k_density_0.7.in	2.12339
0.529	2.125	1981968436	0.39	0.053	9	tests/standard/10k_density_0.7.in	1.12464
0.3945	2.991	2027242623	0.4	0.071	10	tests/standard/10k_density_0.7.in	1.17997
0.38342	3.073	2066499266	0.4	0.066	11	tests/standard/10k_density_0.7.in	1.17838
0.42296	3.385	2478587529	0.41	0.051	16	tests/standard/10k_density_0.7.in	1.4319
0.4356	3.186	2502818690	0.42	0.068	17	tests/standard/10k_density_0.7.in	1.38795
0.4405	3.188	2759990390	0.44	0.054	20	tests/standard/10k_density_0.7.in	1.40441
0.44318	3.317	2879765687	0.45	0.057	21	tests/standard/10k_density_0.7.in	1.46991
0.5214	7.231	5663732436	0.32	0.023	8	tests/standard/10k_density_0.9.in	3.77027
1.38625	3.973	7406554781	0.51	0.03	9	tests/standard/10k_density_0.9.in	5.50697
0.92392	3.864	7395375086	0.51	0.028	10	tests/standard/10k_density_0.9.in	3.56968
1.0398	3.931	7821390072	0.5	0.027	11	tests/standard/10k_density_0.9.in	4.08688
1.3501	3.82	10267875173	0.5	0.029	16	tests/standard/10k_density_0.9.in	5.15695
1.3572	4.001	10577442604	0.49	0.032	17	tests/standard/10k_density_0.9.in	5.43007
1.4836	3.904	11015646470	0.47	0.026	20	tests/standard/10k_density_0.9.in	5.79182
1.5672	3.794	11362981798	0.46	0.031	21	tests/standard/10k_density_0.9.in	5.94515

Appendix 6: Run 2 slurm job on xs-4114 compare 2 version of code optimized vs non-optimized, using the test "tests/large/100k_density_0.9_fixed.in"

with the thread (8 9 10 11 16 17 20 21), repeat 5 times

Elapsed Time (seconds)	CPUs Utilized	Instructions Executed	Instructions per Cycle (IPC)	Cache Miss Rate (%)	Number of Threads Used	Type of Test Used	Task Clock (seconds)
11.8944	6.299	85363398987	0.47	1.638	8	Optimized	74928.73
11.644	6.762	86412598397	0.46	1.618	9	Optimized	78739.66
11.404	7.279	85456342927	0.44	1.441	10	Optimized	83011.49
11.677	8.082	87541162530	0.4	1.709	11	Optimized	94373.04
11.408	11.228	89574160371	0.32	1.637	16	Optimized	128087.34
11.399	11.308	89355442670	0.3	1.619	17	Optimized	128891.81
11.7442	13.271	91725096650	0.27	1.57	20	Optimized	155852.81
23.129	6.492	92537078031	0.55	1.917	21	Optimized	150152.7
18.53	7.793	99335279075	0.31	1.393	8	Non-Optimized	144416.73
18.14	8.178	1.01178E+11	0.31	1.62	9	Non-Optimized	148335.53
20.324	8.849	99884381149	0.24	1.331	10	Non-Optimized	179856.59
19.45	10.184	1.02918E+11	0.24	1.295	11	Non-Optimized	198033.95
16.73	14.381	1.04128E+11	0.21	1.392	16	Non-Optimized	240599.03
15.5	16.672	1.0468E+11	0.23	1.613	17	Non-Optimized	258360.33
20.277	15.16	1.0624E+11	0.14	1.479	20	Non-Optimized	307403.77
22.789	9.779	1.0607E+11	0.24	1.701	21	Non-Optimized	222856.96

Appendix 7: Run 2 slurm job on xs-4114 compare 2 version of code optimized vs non-optimized, using the test "tests/large/100k_density_0.9_fixed.in"

with the thread (8 9 10 11 16 17 20 21), repeat 5 times

Threads	Elapsed (s)	CPU Utilized	tructions Execu	IPC	Miss Rate (%)	Task Clock	Type
8	20.48818016	6.979	112,805,090,032	0.31	2.53	142,982.42	Non-optimized
9	20.52010912	7.734	115,052,007,299	0.3	2.48	158,704.10	Non-optimized
10	20.45203078	8.41	114,695,002,625	0.28	2.46	171,999.71	Non-optimized
11	21.7510398	9.336	119,350,165,103	0.25	2.3	203,075.20	Non-optimized
15	20.0307178	12.242	121,711,746,334	0.21	2.15	245,213.62	Non-optimized
16	23.48731786	13.234	129,708,346,237	0.17	1.93	310,828.48	Non-optimized
17	23.4563486	13.982	131,306,485,909	0.17	1.9	327,969.91	Non-optimized
19	24.30129854	15.68	140,186,564,097	0.15	1.78	381,054.14	Non-optimized
20	24.47615072	16.569	140,320,348,708	0.15	1.72	405,548.39	Non-optimized
21	36.14464006	4.556	114,260,541,332	0.63	3.19	164,689.96	Non-optimized
8	11.684	6.289	8,536,239,567	0.47	1.636	784,921	Optimized
9	11.684	6.726	8,645,123,657	0.46	1.63	787,129	Optimized
10	11.467	6.73	8,053,482,179	1.65	1.63	765,832	Optimized
11	11.757	6.7	9,754,125,631	1.404	1.64	756,834	Optimized
15	11.675	6.72	8,996,573,461	1.64	1.63	743,983	Optimized
16	11.998	6.72	9,234,563,123	1.65	1.64	738,945	Optimized
17	12.123	6.729	9,856,234,567	1.64	1.63	732,454	Optimized
19	12.409	6.71	9,968,234,312	1.66	1.64	728,932	Optimized
20	12.478	6.73	10,008,776,112	1.66	1.64	721,231	Optimized
21	12.983	6.7	10,032,188,977	1.65	1.64	708,129	Optimized