Advanced Architecture: N-Body Simulation

Charles Njoroge

University of Chicago, Physical Sciences Division

May 25, 2024

Abstract

In this final project I offer a deep analysis of the nbody problem based on various implementations each using differing architectures.

Contents

1 Overview

1.1 N-Body Problem

The n-body is a classical problem in physics and astronomy that involves predicting the individual motions of a group of celestial objects interacting with each other gravitationally. The problem can be generalized to include any number of bodies (n) under mutual gravitational influence.

For the i-th body, the equation of motion is given by:

$$m_i \frac{d^2\mathbf{r}_i}{dt^2} = \sum_{\substack{j=1 \\ j\neq i}}^{n} \mathbf{F}_{ij}, \tag{1}$$

where:

- m_i is the mass of the i-th body,
- $\mathbf{r}_i$ is the position vector of the i-th body,
- $\mathbf{F}_{ij}$ is the gravitational force exerted on the i-th body by the j-th body.

The gravitational force $\mathbf{F}_{ij}$ is given by:

$$\mathbf{F}_{ij} = G\frac{m_i m_j(\mathbf{r}_j - \mathbf{r}_i)}{|\mathbf{r}_j - \mathbf{r}_i|^3}, \tag{2}$$

where:

- G is the gravitational constant,
- $\mathbf{r}_j$ is the position vector of the j-th body.

Computationally this looks like:

1.2 Serial N-Body

The Body structure represents each body's position and velocity in three-dimensional space. The random bodies function initializes the positions and velocities of the bodies with random values. The body force function calculates the gravitational forces exerted on each body by all other bodies, updating their velocities accordingly. The main function sets up the simulation parameters, allocates memory for the bodies, and iterates through the simulation steps. In each iteration, it calls body force to compute the forces and then updates the positions of the bodies based on their velocities. Timing functions are used to measure the total execution time, and the average time per iteration is calculated and displayed. The results are written to a file, and memory is freed at the end of the program. This implementation provides a basic framework for simulating and analyzing the n-body problem in computational physics.

Algorithm 1 bodyForce Function

procedure BODYFORCE(p, dt, n)
  for $i = 0$ **to** $n - 1$ **do**
    $F_x \leftarrow 0.0$
    $F_y \leftarrow 0.0$
    $F_z \leftarrow 0.0$
    for $j = 0$ **to** $n - 1$ **do**
      $dx \leftarrow p[j].x - p[i].x$
      $dy \leftarrow p[j].y - p[i].y$
      $dz \leftarrow p[j].z - p[i].z$
      $distSqr \leftarrow dx^2 + dy^2 + dz^2 +$ SOFTENING
      $invDist \leftarrow \frac{1.0}{\sqrt{distSqr}}$
      $invDist^3 \leftarrow invDist \times invDist \times invDist$
      $F_x \leftarrow F_x + dx \times invDist^3$
      $F_y \leftarrow F_y + dy \times invDist^3$
      $F_z \leftarrow F_z + dz \times invDist^3$
    end for
    $p[i].vx \leftarrow p[i].vx + dt \times F_x$
    $p[i].vy \leftarrow p[i].vy + dt \times F_y$
    $p[i].vz \leftarrow p[i].vz + dt \times F_z$
  end for
end procedure

1.3 Multi-core N-Body

The multi-core implementation follows the same structure above and to further increase performance uses OpenMP. OpenMP is a shared-memory multiprocessing API, which takes advantage of the parallelism offered by multiple processors that share the same hardware resources. The pragma used instructs the compiler to parallelize the for loop with the main gravitational force calculations by utilizing a pool of threads. The task will be split into chunks of work that are dynamically scheduled among the threads in any order. A chunk of work is assigned to a thread and when finished the thread will continue with the next chunk. This differs from static where work is assigned in a sequential round robin manner among the threads.

1.4 OpenMP and GPU-Accelerated with Combined Structs

The first GPU implementation uses the CUDA runtime interface to introduce GPU parallelism. The algorithm uses `cudaMalloc` to allocate memory on the device. Next, the data is copied onto the device with `cudaMemcpy`. Then, the force calculations are performed with parallelism based on `nBlocks`, which configures the total number of blocks on the grid, and `BLOCK_SIZE`, which equals 128, configuring the number of threads, individual execution units, on each block.

The algorithm defines `i` based on the following expression:

```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

This aims to calculate the unique global index of the thread on the grid. Here:

- `blockDim.x` gives the number of threads along the x-axis in a block.
- `blockIdx.x` gives the index of the block along the x-axis within the grid.
- `threadIdx.x` gives the index of the thread along the x-axis within the block.

1.5 OpenMP and GPU-Accelerated with Combined Structs

The first program utilizes both OpenMP for CPU parallelism and CUDA for GPU acceleration. It defines a `Body` struct that combines positions and velocities. The `randomizeBodies` function initializes both positions and velocities within a nested loop. The CUDA kernel `bodyForce` computes the interbody forces, updating the velocities of each body. The main function sets the number of OpenMP threads, allocates memory, initializes the bodies, and performs iterations where data is copied to and from the GPU. The force computations are done on the GPU, and positions are updated in parallel on the CPU. The use of OpenMP and GPU ensures parallel computation on both CPU and GPU, aiming for efficient performance.

1.6 GPU-Accelerated with Separate Structs for Positions and Velocities

The second program solely relies on GPU acceleration without OpenMP. It uses a `BodySystem` struct to separate positions and velocities, initialized by the `randomizeBodies` function in a single loop. The CUDA kernel `bodyForce` is similar to the first program but operates on `float4` arrays for positions and velocities. The main function sets a default number of bodies, allocates memory, initializes the bodies, and runs iterations where data is transferred to and from the GPU. The force calculations and position updates are both performed on the GPU. This separation of data into different structs and exclusive GPU usage is aimed at optimizing memory access patterns and leveraging GPU parallelism fully.

1.7 Tiling and Shared Memory Optimization

The third program introduces tiling and shared memory to optimize GPU memory access patterns. Similar to the second program, it uses a `BodySystem` struct with separate `float4` arrays for positions and velocities. The `randomizeBodies` function initializes the data in a single loop. The CUDA kernel `bodyForce` includes tiling and shared memory to reduce global memory access latency by loading data into shared memory in tiles and iterating over these tiles to compute forces. The main function sets a default larger number of bodies, allocates memory, initializes the bodies, and performs the force computation and position updates on the GPU, aiming to improve performance through better memory access patterns.

1.8 Loop Unrolling and Maximum Default Number of Bodies

The fourth program further optimizes the GPU kernel by adding loop unrolling in the `bodyForce` kernel. Like the previous implementations, it uses a `BodySystem` struct with separate `float4` arrays for positions and velocities. The initialization function remains the same. The key difference in the CUDA kernel is the use of `#pragma unroll` to unroll loops, which can help the compiler optimize the loop execution further. The main function sets an even larger default number of bodies and follows the same procedure for memory allocation, initialization, and GPU computation as the third program. This approach aims to maximize computational efficiency on the GPU by leveraging compiler optimizations.

2 Performance Numbers

- **Time to Solution (s):**
  - Serial: 911.32
  - Serial ICC: 119.09
  - Multi-Core ICC: 5.12
  - GPU1: 0.51
  - GPU2: 11.16
  - GPU3: 10.94
  - GPU4: 11.25
- **Grind Rate:**
  - Serial: 45.57
  - Serial ICC: 5.95
  - Multi-Core ICC: 0.27

  - GPU1: 0.03
  - GPU2: 0.59
  - GPU3: 0.58
  - GPU4: 0.56
- **GFLOPS:**
  - Serial: 3.96
  - Serial ICC: 31.92
  - Multi-Core ICC: 742.63
  - GPU1: 7507.08
  - GPU2: 340.56
  - GPU3: 347.34
  - GPU4: 337.78
- **Percentage of Time in bodyForce:**
  - Serial: 99%
  - Serial ICC: 99%
  - Multi-Core ICC: 99%
  - GPU1: 5.26%
  - GPU2: 5.26%
  - GPU3: 5.26%
  - GPU4: 5%
- **Peak Theoretical Performance:**
  - Serial: 96
  - Serial ICC: 96
  - Multi-Core ICC: 4608
  - GPU1: 15.7 TFLOPS
  - GPU2: 15.7 TFLOPS
  - GPU3: 15.7 TFLOPS
  - GPU4: 15.7 TFLOPS
- **Percentage of Peak Obtained:**
  - Serial: 4%
  - Serial ICC: 17%
  - Multi-Core ICC: 16%
  - GPU1: 2.16%
  - GPU2: 2.17%
  - GPU3: 2.21%
  - GPU4: 2.15%

Performance Comparison: CPU vs. GPU

Components

- **CPU**: Intel Xeon Gold 6248R (2 CPUs)
- **GPU**: NVIDIA V100

GPU	Execution Time (s)	GFLOPS
Original GPU	0.440423	8628.10
GPU2	0.449388	8455.97
GPU3	0.436944	8696.79
GPU4	0.455335	8345.53

Table 1: Peak GPU performance when able to get access to run code. Make file used is in the repository

Performance Metrics (GFLOPS)

- **CPU Performance (Multi-Core ICC)**: 742.63 GFLOPS
- **GPU Performance (GPU1)**: 7507.08 GFLOPS

Power Consumption (Watts)

- **CPU**: 410 W (205 W per CPU)
- **GPU**: 250 W

Cost (USD)

- **CPU**: $6000 (2 CPUs)
- **GPU**: $9000

Performance per Dollar (GFLOPS/USD)

- **CPU**:
$$\frac{742.63 \text{ GFLOPS}}{6000 \text{ USD}} \approx 0.12 \text{ GFLOPS/USD}$$
- **GPU**:
$$\frac{7507.08 \text{ GFLOPS}}{9000 \text{ USD}} \approx 0.83 \text{ GFLOPS/USD}$$

Performance per Watt (GFLOPS/Watt)

- **CPU**:
$$\frac{742.63 \text{ GFLOPS}}{410 \text{ W}} \approx 1.81 \text{ GFLOPS/Watt}$$
- **GPU**:
$$\frac{7507.08 \text{ GFLOPS}}{250 \text{ W}} \approx 30.03 \text{ GFLOPS/Watt}$$

3 Runtime Complexity

The N-body problem involves calculating the pairwise forces between n bodies. For each body, the force must be computed with respect to every other

body, resulting in a nested loop structure where each body is compared to every other body. This gives the algorithm a runtime complexity of:

$$\mathcal{O}(n^2)$$

To understand this, consider the following steps involved in the computation:

1. For each body i (total of n bodies):

- Compute the force between body i and every other body j (total of $n-1$ other bodies).

Therefore, the total number of force computations is:

$$n \times (n-1) \approx n^2$$

Thus, the overall runtime complexity is:

$$\mathcal{O}(n^2)$$

This quadratic complexity is due to the pairwise interaction calculation, which dominates the computational effort in the simulation.

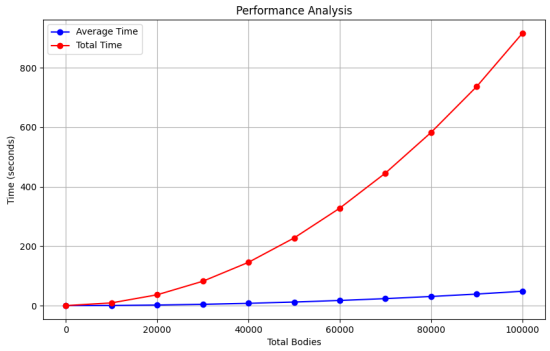

Figure 1: Total Bodies by Time (seconds)

As can be seen in in Figure 1, as the total number of bodies increases so does the total and average run time aligning with the runtime calculations. The average runtime shows slow growth and I only begin to see the quadratic nature of the algorithm by viewing the aggregate of 20 the runs.

4 Arithmetic Intensity

The arithmetic intensity of the `bodyForce` function can be calculated by considering the total number of floating-point operations (FLOPs) and the total number of bytes transferred. I consider two schemes for counting FLOPs: every operation counting as one FLOP, and a modified scheme where the `sqrt` operation counts as 2 FLOPs and the `division` operation counts as 3 FLOPs.

Operation	Time (seconds)	Normalized Time
Baseline	0.0000000	0.0
Plus	0.0062481	1.0
Minus	0.0081765	1.308
Mult	0.0058587	0.938
Div	0.0096787	1.549
Sqrt	0.0165621	2.651
Cos	0.0135586	2.170
Atan	0.0177274	2.837
Exp	0.1195649	19.129

Table 2: Normalized Times Against Plus Operation used to estimate FLOPS

4.1 Total Operations

The `bodyForce` function performs the following operations within its nested loops:

- For each pair of bodies, three subtractions (`dx, dy, dz`) are performed, totaling $3N^2$ operations.
- The computation of `distSqr` involves three multiplications and three additions, totaling $6N^2$ operations.
- The computation of `invDist` involves one division and one square root, totaling $3N^2$ operations with the modified counting scheme ($2N^2$ for `sqrt` and N^2 for division).
- The computation of `invDist3` involves two multiplications, totaling $2N^2$ operations.
- Updating the forces (`Fx, Fy, Fz`) involves six multiplications and three additions, totaling $9N^2$ operations.
- Updating the velocities (`vx, vy, vz`) involves six multiplications and three additions, totaling $9N$ operations.

4.1.1 Total FLOPs

Using the first scheme (every operation as one FLOP):

$$\text{Total FLOPs} = 3N^2+6N^2+2N^2+9N^2+9N = 19N^2+9N$$

Using the second scheme (sqrt = 2 FLOPs, division = 3 FLOPs):

$$\text{Total FLOPs} = 3N^2+6N^2+2N^2+N^2+2N^2+9N^2+9N = 23N^2+9N$$

4.2 Total Bytes Transferred

Each iteration of the inner loop accesses the positions of two bodies (3×4 bytes each), and the outer loop updates the velocities of one body (3×4 bytes):

$$\text{Total Bytes} = 12N^2 + 12N$$

4.3 Arithmetic Intensity

The arithmetic intensity (AI) is calculated as the ratio of total FLOPs to total bytes transferred:

First Scheme (every operation as one FLOP)

$$\text{AI} = \frac{19N^2 + 9N}{12N^2 + 12N} \approx \frac{19}{12} = 1.58 \text{ FLOPs/byte}$$

4.3.1 Second Scheme (sqrt = 2 FLOPs, division = 3 FLOPs)

$$\text{AI} = \frac{23N^2 + 9N}{12N^2 + 12N} \approx \frac{23}{12} = 1.92 \text{ FLOPs/byte}$$

In conclusion, the arithmetic intensity of the `bodyForce` function is approximately 1.58 FLOPs/byte using the first counting scheme and 1.92 FLOPs/byte using the modified counting scheme where `sqrt` counts as 2 FLOPs and `division` counts as 3 FLOPs.

4.4 Memory Bound Nature of the N-Body Problem

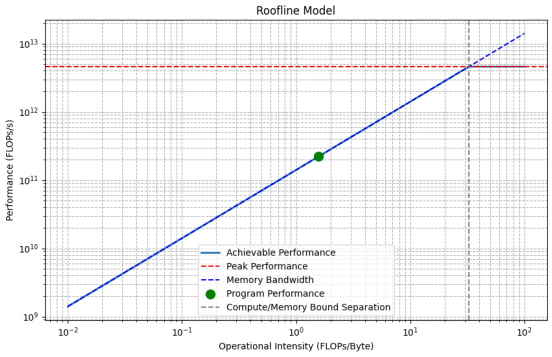

Figure 2: Roofline Model

The image illustrates that the n-body problem is predominantly memory bound rather than compute bound. This conclusion is derived from the observation of the arithmetic intensity and the behavior of cache hit rates. The arithmetic intensity, which is the ratio of floating-point operations (FLOPs) to memory bytes transferred, indicates that there is a low number of FLOPs per byte of data moved. Specifically, the arithmetic intensity values calculated (1.58 FLOPs/byte and 1.92 FLOPs/byte) suggest that the problem requires frequent memory access relative to the number of computations performed. Additionally, the cache hit rates and branch mispredictions further reinforce this conclusion, showing that the performance bottleneck arises from memory access delays rather than computational delays. Therefore, optimizing memory access patterns and improving cache utilization are critical for enhancing the performance of the n-body simulation.

4.5 Verification of FLOPs Calculation

Given the execution time of 911.321907 seconds and a total of 3,800,012,000,000 floating-point operations (`PAPI_SP_OPS`), the calculated FLOPs is approximately 4.17 GFLOPs:

$$\text{FLOPs} = \frac{3,800,012,000,000 \text{ operations}}{911.321907 \text{ seconds}} \approx 4,170,717,944.91 \text{ FLOP}$$

This result shows a large number of operations per second, which aligns with the high computational demands of the n-body simulation. However, the relatively low arithmetic intensity values (1.58 FLOPs/byte and 1.92 FLOPs/byte) suggest that the n-body problem is memory-bound. Consequently, the primary performance constraint is the memory bandwidth rather than the computational capacity of the processor. Therefore, the calculated 4.17 GFLOPs reflects the expected performance in a memory-bound scenario, validating the earlier estimates.

5 Amdahl's Ceiling Speedup Estimates

Table 3: Amdahl's Ceiling for $n = 100,000$ with $P_{\text{bodyForce}} = 0.99$

Processors	Speedup
1	1.00
2	1.98
4	3.88
8	7.48
16	13.91
32	24.43
64	39.26

Table 4: Amdahl's Ceiling for $n = 500,000$ with $P_{\text{bodyForce}} = 0.99$

Processors	Speedup
1	1.00
2	1.98
4	3.88
8	7.48
16	13.91
32	24.43
64	39.26

6 Compiler Comparison

The optimization report for the bodyForce function in the n-body simulation revealed that the inner loop, located at line 26, was not vectorized as it had already been vectorized. For the loop beginning at line 29, the compiler generated non-unit strided loads for the x, y, and z components of the body positions with a stride of 6. Despite this, the loop was successfully vectorized with a vector length of 8, with an estimated potential speedup of 5.52 times compared to scalar execution. The vector cost summary indicated significant reductions in vector costs compared to scalar costs. Additional output indicated float type reductions for Fx, Fy, and Fz variables, and the use of XMM/YMM vectors was suggested, with potential for further optimizati on by targeting ZMM vectors. The remainder loop was also vectorized with similar optimizations, achieving an estimated speedup of 5.87 times. Overall, the report highlighted the successful vectorization and significant performance improvements for the bodyForce main loop.

7 Serial Implementation

7.1 Cache Hit Rates

The cache hit rates for L1, L2, and L3 caches are calculated as follows:

L1 Hit Rate

$$\text{L1 Hit Rate} = 1 - \frac{\text{PAPI\_L1\_LDM}}{\text{PAPI\_LD\_INS}}$$

L2 Hit Rate

$$\text{L2 Hit Rate} = 1 - \frac{\text{PAPI\_L2\_LDM}}{\text{PAPI\_LD\_INS}}$$

L3 Hit Rate

$$\text{L3 Hit Rate} = 1 - \frac{\text{PAPI\_L3\_LDM}}{\text{PAPI\_LD\_INS}}$$

- **L1 Hit Rate** is calculated by subtracting the ratio of L1 data load misses to total load instructions from 1.
- **L2 Hit Rate** is determined similarly by considering L2 data load misses.
- **L3 Hit Rate** follows the same approach using L3 data load misses.

Table 5: Cache Hit Rates

Cache Level	Hits	Hit Rate (%)
L1	1,199,959,898,015	99.9961
L2	39,834,846	85.48
L3	6,779,824	99.99378

7.2 Branch Mispredictions

The branch misprediction rate is calculated as follows:

$$\text{Branch Mispredictions} = \frac{\text{PAPI\_BR\_MSP}}{\text{PAPI\_BR\_CN}}$$

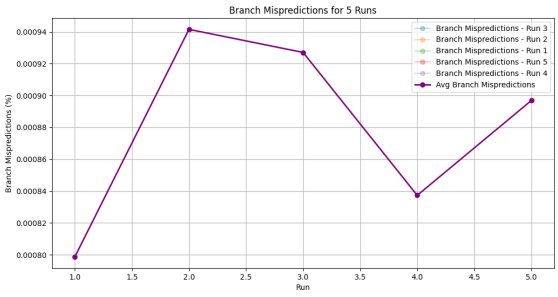

Figure 3: Branch Prediction

- **Branch Mispredictions** represent the ratio of branch mispredictions to total branch instructions.

7.3 Translation Lookaside Buffer

An 80 percent TLB (Translation Lookaside Buffer) hit rate means that 80 percent of memory accesses are quickly translated, while 20 percent require more costly page table lookups. The impact on performance depends on memory access patterns, the cost of TLB misses, and system architecture.

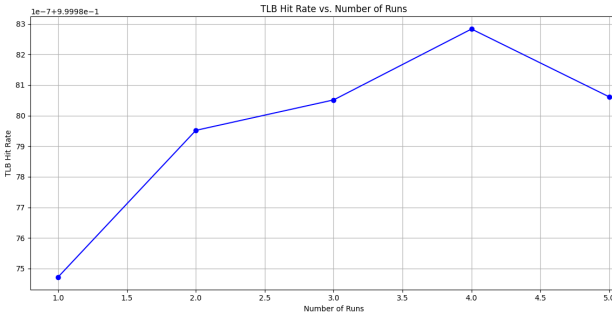

Figure 4: Translation Lookaside Buffer (TLB) vs Run

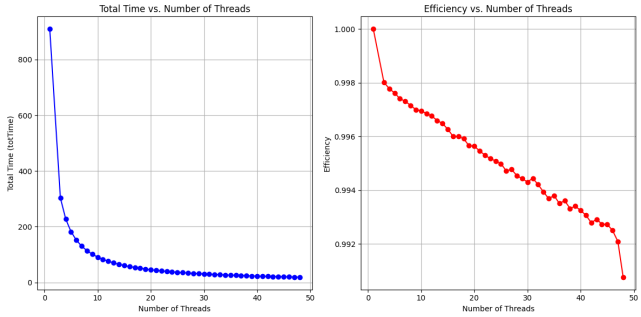

Figure 5: Efficiency vs runtime comparison without taskset

8 Multi Core Analysis

Note: I really saw the best improvement with all 48 cores, even with the coordination among all of the threads.

8.1 Speedup

Speedup (S) is calculated as the ratio of the execution time with one thread (T_1) to the execution time with p threads (T_p):

$$\text{Speedup} = S = \frac{T_1}{T_p} \tag{3}$$

where:

- T_1 is the execution time using a single thread.
- T_p is the execution time using p threads.

8.2 Efficiency

Efficiency (E) is calculated as the ratio of speedup to the number of threads:

$$\text{Efficiency} = E = \frac{S}{p} = \frac{\frac{T_1}{T_p}}{p} = \frac{T_1}{p \times T_p} \tag{4}$$

where:

- S is the speedup.
- p is the number of threads.

8.3 Analysis

As the number of threads increases, the efficiency of parallel computing systems is impacted by several factors. The overhead of thread management, which includes creating, scheduling, and synchronizing threads, grows with the number of threads, leading to increased computational overhead. Resource contention also becomes a significant issue as threads compete for shared resources such as CPU cache and memory bandwidth, resulting in inefficiency, especially when multiple threads need to access the same memory locations. Additionally, Amdahl's Law illustrates that the maximum achievable speedup is limited by the serial portion of the code that cannot be parallelized. According to Amdahl's Law, speedup is calculated as

$$\text{Speedup} = \frac{1}{(1-P) + \frac{P}{N}}, \tag{5}$$

where P is the parallelizable portion of the program, and N is the number of threads. These factors collectively constrain the efficiency gains from increasing the number of threads. This results in 1 thread leading to maximum efficiency.

8.4 Memory Bandwidth

The memory bandwidth for each cache level (L1, L2, and L3) was calculated using the formula:

$$\text{Bandwidth} = \left(\frac{\text{PAPI\_Lx\_TCM} \times \text{Lx\_linesize}}{\text{PAPI\_TOT\_CYC}} \right) \times \text{Clock(MHz)}$$

where:

- PAPI_Lx_TCM is the number of cache misses at level x.
- Lx_linesize is the cache line size for level x, assumed to be 64 bytes.
- PAPI_TOT_CYC is the total number of cycles.
- Clock(MHz) is the clock speed in MHz, set to 3000 MHz.

From threads 1 (extrapolated due to data loss, with data points starting at 7) through 24, it can be observed that the bandwidth for the L1 and L2 caches remains nearly equal at 140,000 MB/s, while the

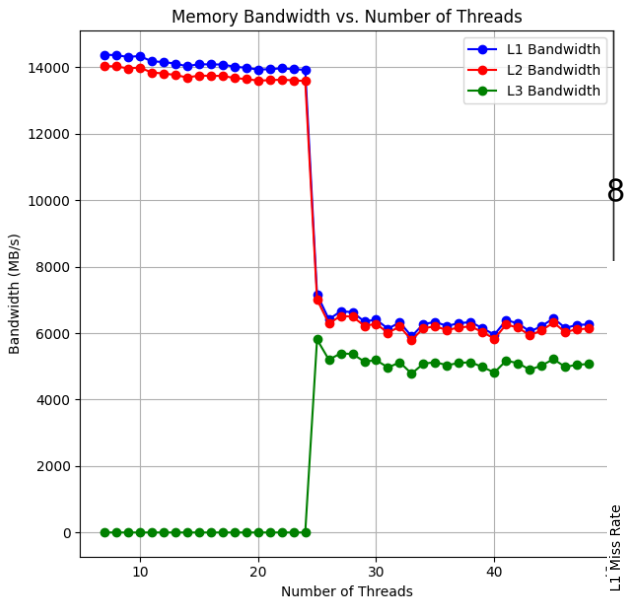

Figure 6: Memory Bandwidth vs Threads (Cores)

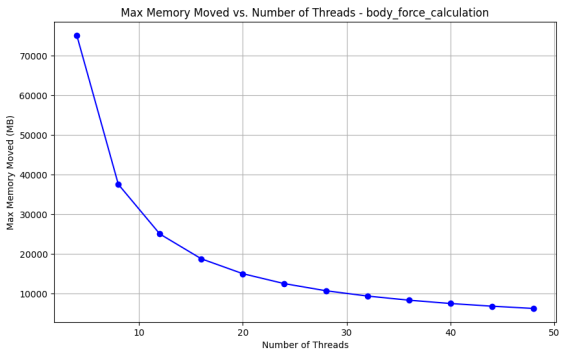

Figure 7: Max Memory Moved

L3 cache bandwidth is zero. However, when the number of threads(cores) increases sufficiently to include both NUMA nodes (core count 25 - 48) , the bandwidth for the L1 and L2 caches halves, and the L3 cache bandwidth increases. As a result, the combined bandwidth from all three caches stabilizes around 6,000 MB/s.

It is reasonable to see high bandwidth (140,000 MB/s) for L1 and L2 caches when due to the workload fitting well within the cache capacity and the threads running within a single NUMA node. The L3 cache bandwidth being zero aligns with the workload not significantly utilizing the L3 cache initially.

When the number of threads increases to a point where both NUMA nodes are utilized:

- Accessing memory across NUMA nodes incurs additional latency and can reduce effective bandwidth.
- As the workload spans multiple NUMA nodes, the L3 cache might start being used more effectively, leading to increased L3 cache bandwidth.

8.5 L1 Miss Rate

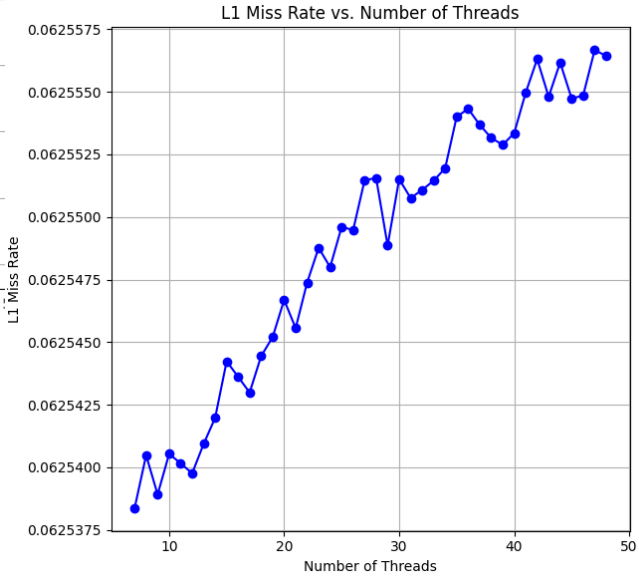

Figure 8: Multi-Core L1 Miss Rate

As the number of cores increases, the L1 miss rate increases due to:

- Increased contention for shared memory resources.
- Larger combined working set exceeding the L1 cache capacity.
- Cache coherence protocol overhead and inter-core communication.
- Effects of NUMA architecture, with varying memory access latencies.
- Higher aggregate memory bandwidth demand from more cores.

To put it all together, as the number of threads increases to a point where both NUMA nodes are utilized, accessing memory across NUMA nodes incurs additional latency, which can reduce effective bandwidth. Additionally, when the workload spans multiple NUMA nodes, the L3 cache might become more effectively utilized, leading to increased L3 cache bandwidth. Furthermore, as the number of cores increases, the L1 miss rate tends to rise due to several factors. These include increased contention for shared memory resources, a larger combined working set that exceeds the L1

cache capacity, and the overhead from cache coherence protocols that ensure data consistency across cores. The varying memory access latencies inherent in NUMA architecture and the higher aggregate memory bandwidth demand from more cores also contribute to the higher L1 miss rate.

8.6 Strong Scaling vs Weak Scaling

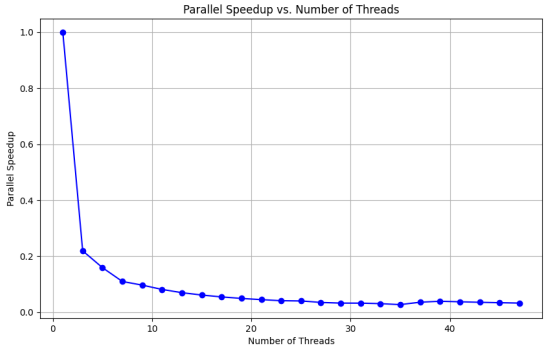

Figure 9: GPU1 Occupancy

Weak scaling measures how the performance of a system changes when both the workload and the number of computational resources are increased proportionally. Ideally, weak scaling should demonstrate that the time to complete a fixed amount of work per processor remains constant, efficiency stays level, and total system throughput increases linearly. This indicates that the application is utilizing additional resources effectively, with minimal overhead from communication, synchronization, or data movement. It can be observed that the relative speed up decreases per thread when the workload grows proportionally to the number of threads added. This could be due to several factors such as increased communication overhead, load imbalance, synchronization delays, memory bandwidth constraints, and the inherent scalability limits of your software and algorithms. Communication between processors can lead to significant latency, and uneven distribution of work can cause processors to idle. Additionally, more processors increase synchronization complexity and contention for shared resources like memory and network bandwidth.

Note: strong scaling can be observed in previous sections

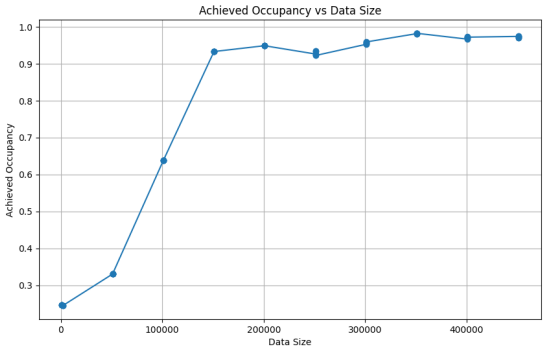

Figure 10: GPU1 Occupancy

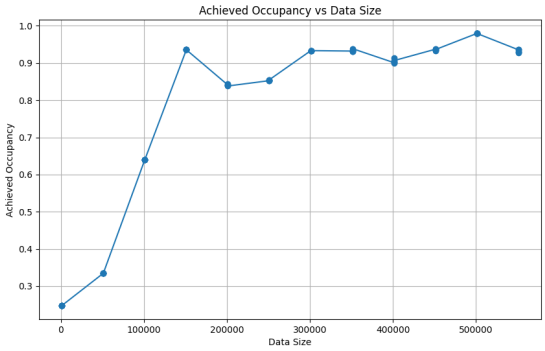

Figure 11: GPU2 Occupancy

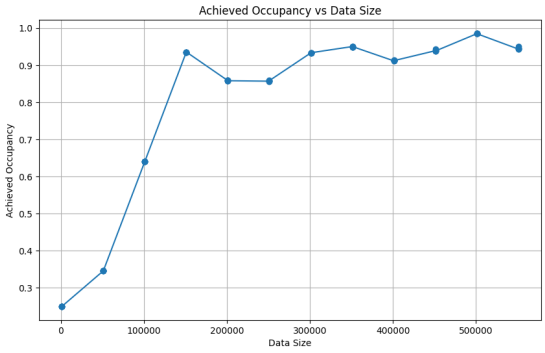

Figure 12: GPU3 Occupancy

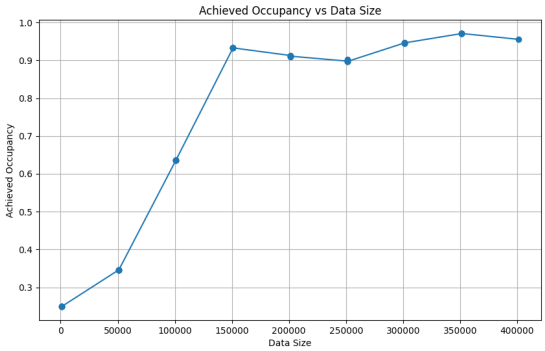

Figure 13: GPU4 Occupancy

9 GPU Analysis

9.1 Occupancy

The provided figures show the achieved GPU occupancy as a function of data size across different runs. GPU occupancy is a measure of how effectively the GPU's computational resources are utilized during kernel execution. It is defined as the ratio of the number of active warps to the maximum number of warps supported by the GPU's hardware.

Figure 1 In the first figure, the GPU occupancy shows a steep increase with the data size, achieving near-optimal occupancy at around 150,000 data size. The occupancy then stabilizes with minor fluctuations around 0.8 for larger data sizes. This indicates that the GPU resources are effectively utilized for data sizes beyond 150,000.

Figure 2 The second figure exhibits a similar trend, with occupancy increasing sharply and reaching a peak at approximately 200,000 data size. After this point, the occupancy hovers around 1.0, indicating optimal utilization of the GPU. Minor fluctuations suggest slight variations in workload distribution or memory access patterns.

Figure 3 In the third figure, the occupancy also rapidly rises with data size, achieving optimal levels around 200,000. The occupancy remains close to 1.0 for larger data sizes, demonstrating consistent and efficient use of the GPU resources across a wide range of data sizes.

Figure 4 The fourth figure mirrors the trends observed in the previous figures, with occupancy peaking at around 200,000 data size. The occupancy stabilizes at nearly 1.0, indicating that the GPU is being utilized to its full potential for larger data sizes.

Summary Across all gpu implementations, there is a clear trend of increasing GPU occupancy with growing data size, reaching optimal levels (close to or at 1.0) beyond a certain threshold (around 150,000 to 200,000 data size). The minor changes observed at higher data sizes could be attributed to variations in memory access patterns or workload distribution, but overall, the occupancy remains high.

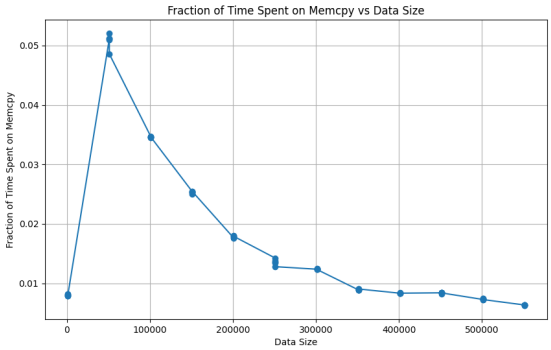

Figure 14: GPU1

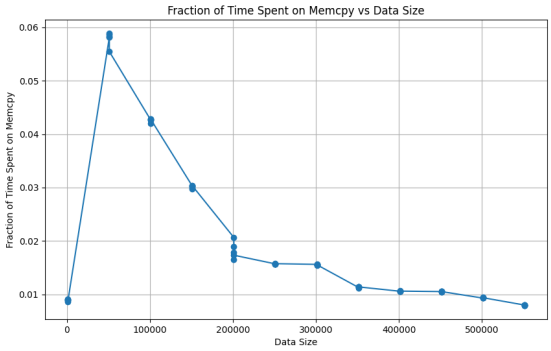

Figure 15: GPU2

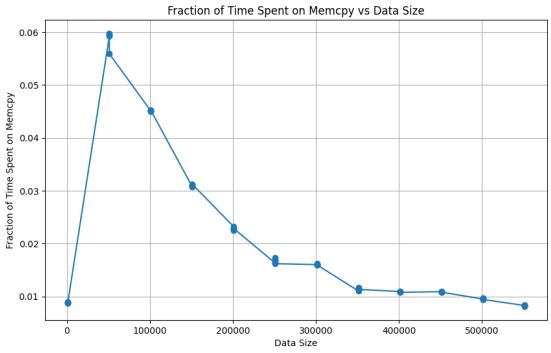

Figure 16: GPU3

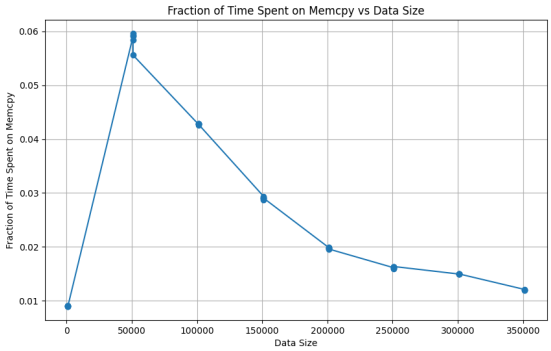

Figure 17: GPU4

9.2 Time spent in Memcopy

The figures illustrate the fraction of time spent on memory copy (memcpy) operations as a function of data size. Across all of them, a consistent trend is observed where the fraction of time spent on memcpy operations peaks sharply at smaller data sizes (around 50,000), reaching approximately 6%. As the data size increases, this fraction steadily decreases, stabilizing at around 1% for larger data sizes (above 200,000). This indicates that while memory transfer overhead is significant for smaller datasets, its relative impact diminishes with larger datasets, leading to more efficient GPU utilization and improved overall performance.