GPU Computing Project Report

David Sevic

January 2025

Abstract

The objective of this project was to implement and optimize an N-body simulation using the Barnes-Hut approximation which reduces computational complexity from $O(N^2)$ to $O(N \log N)$. The project used a quadtree data structure to efficiently group distant bodies and leveraged GPU parallelization techniques to enhance performance. The force computation and position update phases were parallelized, while the quadtree construction remained sequential due to its dynamic nature.

Several optimization strategies were explored, including the use of shared memory to reduce global memory latency and its removal of bank conflicts. Experimental results demonstrated significant performance improvements when using optimized shared memory configurations, with the best runtime achieved using 2048 threads, maintaining the balance between parallelism and memory efficiency.

The findings indicate that while GPU parallelization offers substantial speedup, memory management strategies play a crucial role in achieving optimal performance. This project highlights the effectiveness of GPU acceleration for large-scale simulations and identifies potential areas for further improvement, such as parallel-friendly quadtree construction methods.

Design Methodology

Chosen algorithm

The chosen algorithm for this project is the Barnes-Hut approximation of N-body simulations.

Overview

N-body simulation involves a number of steps, during which, bodies (particles) are moving in a given dimension space, by interacting with all other bodies through gravitational forces. Each body has its own:

- Mass
- Position
- Velocity

Body mass is constant throughout the simulation, while velocities and positions are changed after each simulation step.

Formulas Used

These are the formulas used for N-body simulation:

- **Force**: $\vec{F} = \frac{G \cdot m_1 \cdot m_2}{r^2} \hat{r}$
- **Acceleration**: $\vec{a} = \frac{\vec{F}}{m}$
- **Velocity Update**: $\vec{v}_{t+1} = \vec{v}_t + \vec{a} \cdot \Delta t$
- **Position Update**: $\vec{p}_{t+1} = \vec{p}_t + \vec{v}_{t+1} \cdot \Delta t$

Algorithm Details

The algorithm can be viewed as two parts:

1. Calculation of forces
2. Updating of the bodies (velocities and positions)

The naive approach for calculating forces involves summing up all of the forces from other bodies to the current one, and this approach has the complexity of $\mathcal{O}(N^2)$.

Barnes-Hut approximation is an optimization of the naive approach, where instead of calculating forces between all other bodies, some groups of distant bodies are treated as a single body, and therefore the complexity is reduced to $\mathcal{O}(N \log N)$. The version of the Barnes-Hut approximation in this project uses a quadtree structure in order to represent the bodies. A quadtree is a type of tree that divides a 2D space into quadrants. Nodes are either parent nodes with 4 children nodes, or leaf nodes representing region without further division. The image below illustrates the structure of a quadtree:

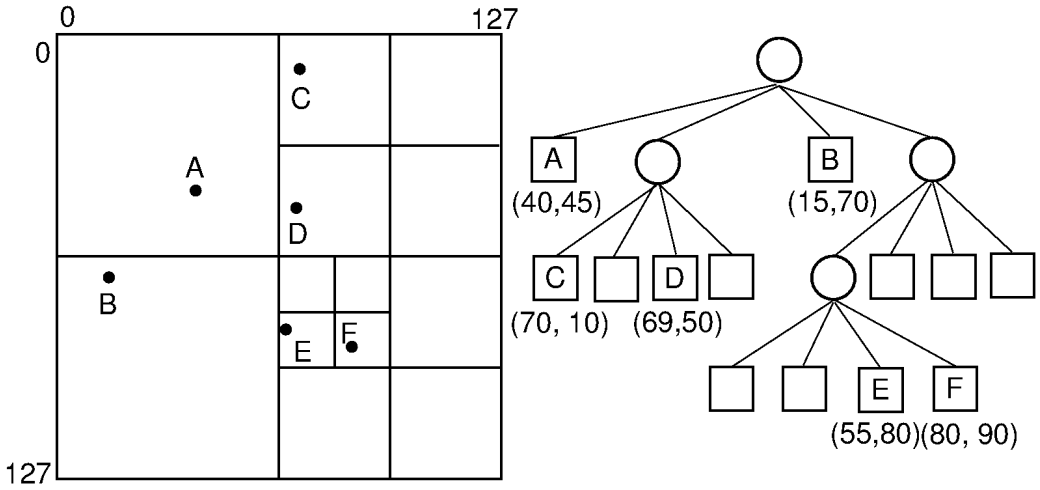

Figure 1: Quadtree structure representing a 2D space

Parallelization Strategy

In the Barnes-Hut approximation, quadtree is used to store bodies into quadrants based on their position in 2D space. When calculating the forces, quadtree is traversed for each body and based on a given threshold parameter, only quadrants which are close enough are divided all the way to the leaf nodes, and those quadrants that are distant enough, are treated as a single body, therefore reducing complexity.

The creation of a quadtree can be performed in parallel, however for this project, this part is left sequential, which does affect the overall results (more about this in the results section).

The force calculations are applied to each body independently and this is where parallelism can be efficiently utilized. After calculating the forces, the updating of the bodies' velocities and positions is also done independently for each body and is therefore also perfectly parallelizable.

The code used for quadtree creation and traversing is created by taking motivation from available pseudo code: patterns.eecs.berkeley.edu.

Quadtree Implementation

The quadtree in this project is represented as an array in memory, consisted of quadrants.

```
const int QUADRANT_SIZE = 12;
const int CHILDREN_0 = 0;
const int CHILDREN_1 = 1;
const int CHILDREN_2 = 2;
const int CHILDREN_3 = 3;
const int CENTER_OF_MASS_X = 4;
const int CENTER_OF_MASS_Y = 5;
const int TOTAL_MASS = 6;
const int X_MIN = 7;
const int X_MAX = 8;
const int Y_MIN = 9;
const int Y_MAX = 10;
const int PARTICLE_INDEX = 11;

const double THETA = 5e-1;
const int QUADTREE_MAX_DEPTH = 10;
const int QUADTREE_MAX_SIZE = static_cast<int>(pow(4, QUADTREE_MAX_DEPTH));

using Quadrant = std::array<double, QUADRANT_SIZE>;
std::vector<Quadrant> quadtree;
```

Each quadrant is consisted of 12 different fields:

- 4 fields for children quadrants
- X and Y coordinates of quadrant's center of mass
- Total mass of the quadrant
- Min and Max coordinates of the quadrant
- Index of the particle (body) in the quadrant

For increased precision, we are using type `double`. However, `float` could be used, and therefore half the size of the quadtree in memory.

Initially, for the CPU implementation, the quadtree did not have the maximum depth limit. However, when we started with the GPU implementation and usage of shared memory, we had to limit the maximum size of

the quadtree in order to fit into the available hardware memory. Because of this limit, it is possible, and quite often the case, that multiple particles will end up in the same leaf node at maximum depth, which does not divide further. Implementation-wise, this requested some changes in the initial logic, and algorithm-wise this does not present an issue, because this type of aggregation is already happening in case of distant quadrants.

The other predefined limit `QUADTREE_MAX_SIZE` presents a practical maximum number of quadrants in the quadtree and is set to:

$$\frac{4^{\text{QUADTREE\_MAX\_DEPTH}} - 1}{3}.$$

The reason for this is that in the worst-case scenario, when all of the bodies are distant enough, at each level of the tree, there will be particles in all 4 child nodes. This calculation accounts for the total number of quadrants in a balanced quadtree by summing the number of nodes at each level of depth. The division by 3 comes from the formula for the sum of a geometric series:

$$\sum_{i=0}^{d} 4^i = \frac{4^{d+1} - 1}{3},$$

which accurately represents the total number of quadrants in the tree across all levels.

In our project, this is the case in the initial and possibly few early stages, because initially, the positions of the bodies are random and evenly distributed over the space. In the later simulation steps, due to some bodies being heavier, the lighter bodies will move towards them, and eventually, the tree becomes sparse, having fewer quadrants.

For the given values of a max depth of 10, the maximum number of quadrants is:

$$\frac{4^{11} - 1}{3}.$$

With each quadrant having 12 `double` values, the maximum memory of the quadtree is around 128 MB, which is way bigger than the available shared memory. Therefore, depending on the number of bodies, the quadtree might not fit into shared memory in one or a few early simulation steps, until it becomes sparse enough.

Quadtree Visualization and Improvements

The first image shows the quadtree in the initial state, where all the bodies are evenly spread out across the quadrants. The second image shows the quadtree in the last simulation step, where most of the bodies have been drawn to the heaviest ones, and therefore the quadtree has much fewer quadrants. The red dots represent quadrants with at least one body inside. We say at least, because after adding the max depth limit, some quadrants end up having multiple bodies inside of them, and since we don't need information about which particles are inside of these max depth quadrants, in order to save memory, we can only present them as one red dot.

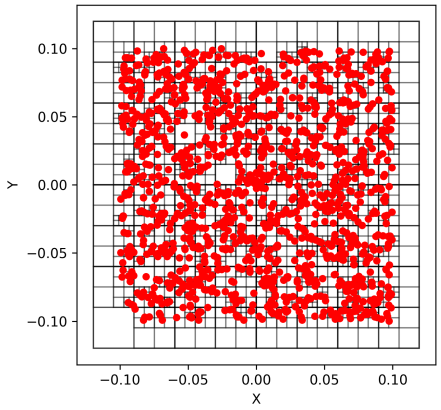

Figure 2: Example of a quadtree at its initial step of the simulation

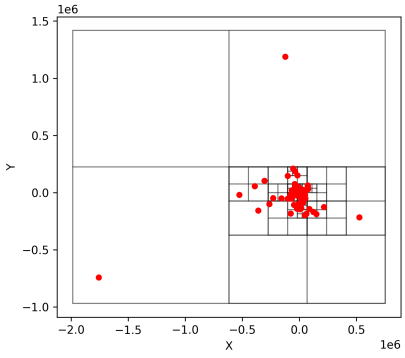

Figure 3: Example of a quadtree at its final step of the simulation

Quadtree Initialization and Body Insertion

The quadtree is initially created with a root quadrant, covering the whole 2D space, which is used to initialize the positions of the bodies. After that, each body is inserted in a sequence. The insertion of bodies is complex, and it consists of:

- Finding the quadrant in which the inserting body belongs.
- If this quadrant is already occupied, dividing it into 4 children nodes (if less than the max allowed depth).
- Reassigning the previous leaf node body as one of the children.

Challenges in Parallel Body Insertion

Because of this dynamic approach, if bodies were to be inserted in parallel, it would cause conflicts in situations where the leaf nodes are divided, and bodies are reassigned. However, there are other approaches which are parallel-friendly, such as encoding bodies with Morton code, sorting them bodies based on their codes and creating subtrees at different levels in parallel. This is one of the ways in which this project could be improved.

Quadtree Construction Function

The following function builds the quadtree using the positions and masses of the bodies:

```
std::vector<Quadrant> buildTree(const Positions& positions, const Masses& masses) {
    quadtree.clear();

    std::array<double, 4> rootBounds = ComputeRootBounds(positions);
    double xMin = rootBounds[0];
    double xMax = rootBounds[1];
    double yMin = rootBounds[2];
    double yMax = rootBounds[3];

    InitializeRoot(xMin, xMax, yMin, yMax);

    for (int i = 0; i < N_BODIES; ++i) {
        QuadInsert(i, 0, positions, masses, 1);
    }
    ComputeMass(0);
    return quadtree;
}
```

Post-Construction Computation

After creating the quadtree, one additional traversal is performed to compute the center and total mass of each quadrant. There is an alternative approach where this computation can be done on the fly while traversing the bodies during the force computation part. This alternative would increase time complexity but save memory space for storing this information.

In this project, we held onto the motivational CPU implementation, where it was done in this way.

Simulation Run

The N-body simulation consists of a number of simulation steps which are performed in a loop. In each iteration step, we are doing the following:

```
loop:
    create_quadtree
    allocate_quadtree_mem_on_gpu
    copy_quadtree_from_cpu_to_gpu
    compute_forces_gpu
    compute_acc_vel_pos_gpu
    copy_positions_from_gpu_to_cpu
    deallocate_quadtree_mem_on_gpu
```

This pseudo code explains the logic of the simulation step loop, which will later be shown and explained in code. Before defining the loop in the code, it is necessary to define and allocate memory.

```
// declaration of GPU memory
double* masses_d;
double* positions_d;
double* velocities_d;
double* accelerations_d;
double* forces_d;
double* quadtree_d;

// allocating GPU memory
cudaMalloc((void**)&masses_d, N_BODIES * sizeof(double));
cudaMalloc((void**)&positions_d, N_BODIES * N_DIM * sizeof(double));
cudaMalloc((void**)&velocities_d, N_BODIES * N_DIM * sizeof(double));
cudaMalloc((void**)&accelerations_d, N_BODIES * N_DIM * sizeof(double));
cudaMalloc((void**)&forces_d, N_BODIES * N_DIM * sizeof(double));
```

In this implementation, we are working with arrays of type `double` for body masses, positions, velocities, and forces. Out of these, only masses contain one value per body, while the other ones contain `N_DIM` values (which is always 2 in our case, since we are using a quadtree optimization; if an octree is used, then `N_DIM` is equal to 3). Based on this, we are allocating the required amount of memory on the GPU.

Memory Allocation Approaches

Since the quadtree potentially changes its size in each iteration, we have to decide how to allocate its memory on the GPU. Two approaches are considered:

1. **Per-Iteration Allocation:** Memory is allocated at the start of each iteration, after building the tree on the CPU, based on its exact size, and then deallocated at the end of the iteration. This approach:
    - Allocates only the exact amount of required memory, avoiding unnecessary use of global GPU memory.
    - Introduces allocation and deallocation in each step of the simulation, creating overhead.
2. **Single Allocation:** Memory is allocated once before entering the loop of simulation steps and deallocated after the loop. This approach:
    - Avoids the overhead of multiple allocations and deallocations.
    - Requires allocating the maximum expected amount of memory for the quadtree, leaving part of it unused as the simulation progresses.

In this implementation, the bodies are initialized with random masses. As the simulation steps advance, lighter bodies are drawn to heavier ones, making the quadtree sparse and reducing its size. The quadtree is likely to reach its maximum size at the start, with each iteration causing it to shrink. However, with the single allocation approach, the maximum size remains allocated on the GPU, and part of it will remain unused.

Comparison of Memory Allocation Approaches

Both approaches have their upsides and downsides. After executing the program with both approaches and different parameters, we measured nearly identical run times. With a larger number of simulation steps, the overhead of the first approach accumulates more. However, it makes up only a scriptsize fraction of the total time, which is similar to the second approach. Therefore, we decided to use the second approach and avoid multiple allocations and deallocations of GPU memory, as it is generally not a good practice.

```
// allocation of maximum size quadtree
size_t realistic_size = std::min(4 * N_BODIES, QUADTREE_MAX_SIZE);
cudaMalloc((void**)&quadtree_d, realistic_size * sizeof(Quadrant));
```

At the beginning of the simulation, the positions of bodies are random and uniformly spread over the space. Because of that, the tree will be balanced, with a tendency for the bodies to end up as leaves of the quadtree. Therefore, we calculate the upper limit by multiplying the number of bodies by the branching factor of 4. Because of the nature of this algorithm, after the first iteration, the lighter bodies move towards the heavier ones, making the tree sparse and smaller in size. Also, because we have the limit of maximum depth, we must ensure not to allocate more than the allowed maximum size of the quadtree (hence the use of the `min` function).

At the beginning, each body is initialized with its mass, starting position, and starting velocity. Accelerations and forces are not initialized as they are calculated later. Therefore, we have to copy the initialized values from the CPU to the GPU's allocated memory.

```
// copying initial values to GPU
cudaMemcpy(masses_d, masses.data(), N_BODIES * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(positions_d, positions.data(), N_BODIES * N_DIM * sizeof(double),
↪  cudaMemcpyHostToDevice);
cudaMemcpy(velocities_d, velocities.data(), N_BODIES * N_DIM * sizeof(double),
↪  cudaMemcpyHostToDevice);
```

After this setup, we are ready to define the simulation loop. As our formulas take time steps into account, we keep track of the absolute time passed and increase it at each iteration by the defined time step.

```
const double DELTA_T = 1.0;
...
for (int step = 0; step < N_SIMULATIONS; ++step) {
    absolute_t += DELTA_T;
    ...
```

At the beginning of each iteration, the first step is the creation of the quadtree. As already mentioned, this computation is performed on the CPU. Therefore, after creating it, the quadtree has to be copied to the GPU.

Copying Quadtree to GPU and Using Shared Memory

```
// copy tree to GPU
cudaMemcpy(quadtree_d, quadtree.data(), quadtree.size() * sizeof(Quadrant),
↪  cudaMemcpyHostToDevice);

// calculate the quadtree size
int quadtreeMemSize = quadtree.size() * sizeof(Quadrant);

// use shared memory if tree can fit in it
int sharedMemSize = quadtreeMemSize <= MAX_SHARED_MEM_PER_BLOCK_B ? quadtreeMemSize : 0;
```

As the quadtree is shared amongst all threads and frequently accessed, we can benefit from storing it in shared memory. Since its size changes in each iteration, we need to calculate whether it can fit into shared memory or not.

We compare its size with the maximum available shared memory, which is 48KB for our hardware. This calculated memory size will later be passed to the kernel.

Block Dimension and Memory Usage

For the block dimension, we decided to use 1D since all our data is composed of 1D arrays. The block size can vary depending on whether shared memory is used or not.

Global Memory Case When using global memory, each SM (Streaming Multiprocessor) is assigned multiple blocks based on available resources. Smaller block sizes cause each SM to be assigned more blocks, which increases its occupancy and can improve performance.

Shared Memory Case When using shared memory, the number of blocks per SM is restricted by the size of allocated shared memory. Since the per-block limit is 48KB and the per-SM limit is 64KB, if we allocate, for example, 40KB, only one block will be stored per SM. Therefore:

- When using shared memory, it is beneficial to use larger block sizes to make the most out of the SM's resources.

Optimal Block Size Calculation

Following this logic, we designed a helper function to return the optimal block size based on two factors:

1. **Hardware limits of the used GPU**, provided using the `cudaDeviceProperties()` function from `cuda_runtime.h`:
   - Device Name: NVIDIA T600
   - Total Global Memory: 3696 MB
   - Max Threads per Block: 1024
   - Max Threads per SM: 1024
   - Max Blocks per SM: 16
   - Warp Size: 32
   - Registers per SM: 65536
   - Shared Memory per Block: 49152 bytes
   - Shared Memory per SM: 65536 bytes
   - Max Warps per SM: 32
2. **Number of registers used in the kernel function**, obtained using the `--ptxas-options=-v` flag when compiling the code:

```
ptxas info    : Compiling entry function '_Z16computeForcesGpuPdS_S_S_ib
ptxas info    : Function properties for _Z16computeForcesGpuPdS_S_S_ib
ptxas info    : Used 70 registers, used 1 barriers, 152 bytes cumulative
                stack size, 357 bytes cmem[0], 64 bytes cmem[2]
```

When observing the kernel functions, we can notice that there are far fewer local variables than reported used registers. The reason is that, beyond local variables, registers are used for storing temporary results, implicit variables such as `threadIdx` and `blockIdx`, and compiler optimizations.

```
// calculate blocksize
int blockSize = getOptimalBlockSize(FIRST_KERNEL_REGISTERS_NUM, sharedMemSize);
```

The helper function takes the number of registers used and the amount of shared memory for the kernel function. The number of registers is measured during compile time, so it does not change during execution. Therefore, we define it as a constant. If any changes are made to the kernel function, the number of used registers should be reevaluated and redefined. Using these two arguments, along with the GPU properties shown above, the function calculates the optimal block size to maximize the SM resources.

```
int getOptimalBlockSize(int registersPerThread, int sharedMemPerBlock = 0, bool
↪  print=false) {
    // how many blocks per SM by shared memory limit
    int blocksPerSM_sharedMem = maxSharedMemPerSM / (sharedMemPerBlock > 0 ?
    ↪  sharedMemPerBlock : 1);
    // how many threads can fit by register usage
    int threadsPerSM_registers = registersPerSM / registersPerThread;
    // effective blocks per SM
    int effectiveBlocksPerSM = std::min(blocksPerSM_sharedMem, maxBlocksPerSM);
    // effective threads per SM
    int effectiveThreadsPerSM = std::min(threadsPerSM_registers, maxThreadsPerSM);
    // effective optimal block size
    int optimalBlockSize = effectiveThreadsPerSM / effectiveBlocksPerSM;
    // round up to be a multiple of warp size
    optimalBlockSize = (optimalBlockSize / threadsPerWarp) * threadsPerWarp;
    // threads per block limit
    optimalBlockSize = std::min(optimalBlockSize, threadsPerBlockLimit);
    return optimalBlockSize;
}
```

Parametrizing the Number of Threads

Initially, this implementation used the same number of threads as the number of bodies in the simulation. However, to measure performance in more detail, we changed it so that the number of threads is parametrized. Therefore, the number of blocks is calculated considering the block size and number of threads.

```
// defining dimensions
dim3 dimBlock(blockSize);
// depends on N_THREADS for arbitrary number of threads approach
dim3 dimGrid((N_THREADS + blockSize - 1) / blockSize);
```

Execution of the kernel code for force computation is defined by passing the calculated number and dimension of blocks, and size of shared memory. Amongst other parameters, we are passing a boolean conditional parameter `sharedMemSize > 0`, which indicates whether or not to work with shared memory. After the kernel execution, we synchronize the CPU code, because we need the computed forces to proceed with the algorithm.

```
// pass quadtree memory size for dynamic allocation of shared memory
computeForcesGpu<<<dimGrid, dimBlock, sharedMemSize>>>(positions_d,
       masses_d, forces_d, quadtree_d, quadtree.size(), sharedMemSize > 0);
cudaDeviceSynchronize();
```

Kernel Code for Force Computation

The kernel code for computation of forces starts with the declaration of shared memory and the calculation of the global thread ID.

```
__global__ void computeForcesGpu(double* positions, double* masses, double* forces,
                                 double* globalMemQuadtree, int quadtreeNumElem, bool
                                 ↪  useSharedMem) {
    // declaration of shared memory for quadtree
    extern __shared__ double sharedMemQuadtree[];

    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N_BODIES)
        return;
```

In order to proceed with the calculation of forces, we first need to store the quadtree from global to shared memory. This can be done in parallel by assigning different quadrants based on thread IDs.

```
if (useSharedMem) {
    for (int i = threadIdx.x; i < quadtreeNumElem; i += blockDim.x) {
        sharedMemQuadtree[i * QUADRANT_SIZE + CHILDREN_0] =
            globalMemQuadtree[i * QUADRANT_SIZE + CHILDREN_0];
        sharedMemQuadtree[i * QUADRANT_SIZE + CHILDREN_1] =
            globalMemQuadtree[i * QUADRANT_SIZE + CHILDREN_1];
        ...
    }
    syncthreads();
}
```

All threads of a block enter this piece of code and are responsible for transferring one quadrant each, from global to shared memory, in each iteration. For example, if we have a block size of 64 and 128 quadrants in the tree:

- In the first iteration, the 0th thread transfers the 0th quadrant, the 1st thread the 1st quadrant, and the 63rd thread the 63rd quadrant.
- In the second iteration, the 0th thread transfers the 64th quadrant, the 1st thread the 65th quadrant, and the 63rd thread the 127th quadrant.

It is crucial to call `syncthreads()` after the loop, so that no thread can advance until the whole quadtree has been transferred to shared memory.

Addressing Bank Conflicts in Shared Memory

After introducing shared memory with this approach, the performance did not improve at all. After analyzing, we realized that a large number of bank conflicts were happening in this approach.

If we look at the thread level, in order to transfer one quadrant from global to shared memory, each thread copies all 12 fields of a quadrant. These fields are stored sequentially in memory, meaning:

- The 0th thread accesses banks 0–11 for transferring one quadrant.
- The 1st thread accesses banks 12–23.
- The 3rd thread accesses banks 24–31 and 0–3, and so on.

With 32 threads in a warp executing simultaneously, this causes a lot of bank conflicts, affecting performance and canceling out the advantages of shared memory.

Resolving Bank Conflicts To resolve this, we can store quadrant fields differently in shared memory. Since these fields are independent and accessed by index, it is not a problem if they are not placed sequentially in memory. Instead:

- Each thread should access only one bank in shared memory.
- For 32 banks, the 0th thread should store all 12 quadrant fields into the 0th bank, the 1st thread into the 1st bank, and the 31st thread into the 31st bank.

To achieve this, we store the fields with an offset equal to the number of banks (32 in our case). This ensures each thread in a warp accesses a unique bank, eliminating conflicts.

```
if (useSharedMem) {
    for (int i = threadIdx.x; i < quadtreeNumElem; i += blockDim.x) {
        sharedMemQuadtree[i * SHARED_MEM_BANKS_NUM + CHILDREN_0] =
            globalMemQuadtree[i * QUADRANT_SIZE + CHILDREN_0];

        sharedMemQuadtree[i * SHARED_MEM_BANKS_NUM + CHILDREN_1] =
            globalMemQuadtree[i * QUADRANT_SIZE + CHILDREN_1];
        ...
    }
    syncthreads();
}
```

It is true that if we use a block size larger than 32, multiple threads would access the same banks. However, it is important to remember that separate warps are usually not executed simultaneously, and therefore do not cause conflicts.

This approach removes the write bank conflict because we can manually determine which locations are accessed by threads based on their index. Later in the kernel code, threads will read from this shared memory unpredictably. Since each thread accesses different quadtree nodes based on the location of the body it is processing, there will likely be read bank conflicts, which are difficult to eliminate.

Optimizing Global Memory Access

While we have optimized shared memory access during this data transfer, we must also consider global memory. Global memory does not have banks, but coalesced access is critical for minimizing latency. Threads should access consecutive memory locations relative to each other to allow the GPU to combine transactions efficiently.

In the current code:

- The 0th thread accesses locations 0–11,
- The 1st thread accesses locations 12–23, and so on.

To make this access coalesced:

- The 0th thread should access location 0, the 1st thread location 1, and so on.

```
for (int i = threadIdx.x; i < quadtreeNumElem * QUADRANT_SIZE; i += blockDim.x) {
    double val = globalMemQuadtree[i];

    int quadrantIdx = i / QUADRANT_SIZE;
    int fieldIdx    = i % QUADRANT_SIZE;

    sharedMemQuadtree[quadrantIdx + SHARED_MEM_BANKS_NUM * fieldIdx] = val;
}
```

This loop modifies the access pattern so that one field is accessed at a time from global memory. For each field, the code:

- Calculates the quadrant and field it belongs to,
- Inserts it into the appropriate location in shared memory.

This ensures that threads access global memory locations sequentially, achieving coalesced access. Local variables are shown here for readability but were removed in the actual code.

Parallelizing Force Calculation with Quadtree Traversal

As already mentioned, in the Barnes-Hut optimization, the calculation of forces is done by traversing the quadtree. This is performed for each body independently, making it highly parallelizable as it can be executed simultaneously. The straightforward approach involves assigning threads to be responsible for force calculation over one or more bodies, depending on the problem size and number of threads.

Tree Traversal for Force Calculation

The tree traversal is implemented as a Depth First Search (DFS) algorithm, which uses a stack structure. Initially, the root node is pushed onto the stack, and a loop is entered. Within the loop:

- A node is popped from the stack and checked to determine whether it should be divided further.
- If the node is a leaf or is distant enough based on the approximation parameter:
  - It is not divided further.
  - The force between the node and the current body being processed is calculated.
- If the node is close enough:
  - It is divided further by pushing all four of its child nodes onto the stack.

This process is repeated until the stack is empty and the traversal is complete.

The following code snippet demonstrates the repeated traversal for multiple bodies:

```
// repeat for multiple bodies
for (int body_i = idx; body_i < N_BODIES; body_i += N_THREADS) {
    double sum[2] = {0, 0};

    double pos_i[2] = {positions[body_i * 2 + 0], positions[body_i * 2 + 1]};

    int nodeStack[QUADTREE_MAX_DEPTH * 3 + 1];
    int stack_top = -1;

    // push the root node (rootIndex is 0)
    if (!push(nodeStack, &stack_top, 0)) {
        printf("Stack overflow while pushing root node.\n");
        return;
    }
    ...
}
```

Stack Implementation for Quadtree Traversal

The stack is implemented as an integer array, as we are storing node indices in it. Beyond storing in shared memory, another reason for limiting the maximum depth of the quadtree was to set an upper limit on the stack size in kernel code. Since this stack is a local variable, it can be stored in the register memory of the thread, which provides the fastest access. However, if the stack is too large, it might be placed into global memory, which is significantly slower and lowers performance.

Alternative Memory Considerations There was a consideration to store this stack in a different type of memory, such as shared memory. However, since every thread has a different traversal path, this approach would not make sense.

Stack Size Limit The stack size is limited to

$$\text{QUADTREE\_MAX\_DEPTH} \times 3 + 1$$

. This limit is derived from the fact that during traversal:

- At each level, one node is popped.
- A maximum of four nodes is pushed, resulting in a net gain of three nodes per level.

The additional `+1` accounts for the root node.

Stack Operations in Kernel Code

The stack's push and pop operations are implemented as separate functions using the `__device__` qualifier to indicate that they are used in kernel code.

```
__device__ bool push(int stack[], int *top, int value) {
    if (*top >= QUADTREE_MAX_DEPTH * 3) {
        // stack overflow
        return false;
    }
    stack[++(*top)] = value;
    return true;
}

__device__ bool pop(int stack[], int *top, int *value) {
    if (*top < 0) {
        // stack underflow
        return false;
    }
    *value = stack[(*top)--];
    return true;
}
```

Outer Loop for Body Distribution

The outer loop ensures that bodies are distributed across threads correctly. This allows threads to process multiple bodies if there are fewer threads than bodies. Additionally, we ensure that the quadtree is accessed with the correct indexing, depending on whether shared memory is used.

```
for (int body_i = idx; body_i < N_BODIES; body_i += N_THREADS) {
    ...

    while (stack_top >= 0) {
        int nodeIndex;
        if (!pop(nodeStack, &stack_top, &nodeIndex)) {
            printf("Stack underflow while popping node.\n");
            break;
        }

        double* node = useSharedMem
            ? &quadtree[nodeIndex]
            : &quadtree[nodeIndex * QUADRANT_SIZE];

        const int ACCESS_INDEX = useSharedMem ? SHARED_MEM_BANKS_NUM : 1;

        ...
    }
}
```

Optimizing Kernel Code for Force Computation

The rest of the force computation logic follows the approach we have already explained. However, we focused on one more way to optimize the kernel code. As mentioned earlier, the local variables in the kernel code are first attempted to be stored in the thread registers. The total number of registers is limited per SM, and by reducing the number of registers used, we allow more threads to execute on an SM simultaneously, increasing occupancy and overall performance.

Reviewing Local Variables

It is useful to examine the local variable definitions in the kernel code to determine if they are necessary:

```
double pos_i[2] = {positions[body_i * 2 + 0], positions[body_i * 2 + 1]};
...
displacement[0] = node[ACCESS_INDEX * CENTER_OF_MASS_X] - pos_i[0];
displacement[1] = node[ACCESS_INDEX * CENTER_OF_MASS_Y] - pos_i[1];
...
```

Some variables, such as the `pos_i` array, were initially used simply for code readability and were referenced only once after their definition. These local variables should definitely be removed, as they provide no computational benefits and only consume the limited registers.

```
double sum[2] = {0, 0};
...
while (stack_top >= 0) {
    ...
    sum[0] += force_mag * nx;
    sum[1] += force_mag * ny;
    ...
}
forces[body_i * 2 + 0] = sum[0];
forces[body_i * 2 + 1] = sum[1];
```

On the other hand, local variables such as the `sum` array should not be removed. While it is true that it can be replaced with direct accesses to the `forces` array, freeing up registers:

```
forces[body_i * 2 + 0] = 0;
forces[body_i * 2 + 1] = 0;
...
while (stack_top >= 0) {
    ...
    forces[body_i * 2 + 0] += force_mag * nx;
    forces[body_i * 2 + 1] += force_mag * ny;
    ...
}
```

The `forces` array is stored in global memory. For each force calculation pair, this array would be accessed with two reads and two writes. Even though the Barnes-Hut optimization reduces the number of pairings, this still results in a significant number of global memory accesses. On the other hand, when using the `sum` array, we have only 2 writes at the end of the force calculations.

Alternative Considerations We could insist on minimizing the number of used registers, and going with the second approach, but then we would have to consider allocating shared memory for intermediate results and storing only the final values into the global `forces` array. However, since we are already taking up most and sometimes all of the available shared memory for the quadtree, we stuck with the usage of registers in situations like this.

Updating Positions of Bodies and Finalizing the Simulation

```
// defining dimensions for rest of the calculation
blockSize = 64;
dimBlock = dim3(blockSize);

// depends on N_THREADS for arbitrary number of threads approach
dimGrid = dim3((N_THREADS + blockSize - 1) / blockSize);

updateAccVelPos<<<dimGrid, dimBlock>>>(forces_d, masses_d, accelerations_d,
                                       velocities_d, positions_d, DELTA_T);
cudaDeviceSynchronize();
```

After calculating the forces, the next part of the simulation step is updating the positions of the bodies. This involves several computations and is grouped into a single kernel code:

```
__global__ void updateAccVelPos(double* forces, double* masses, double*
                                accelerations, double* velocities,
                                double* positions, double DELTA_T) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx >= N_BODIES)
        return;

    for (int body_i = idx; body_i < N_BODIES; body_i += N_THREADS) {
        accelerations[body_i * 2 + 0] = forces[body_i * 2 + 0] / masses[body_i];
        accelerations[body_i * 2 + 1] = forces[body_i * 2 + 1] / masses[body_i];

        velocities[body_i * 2 + 0] += accelerations[body_i * 2 + 0] * DELTA_T;
        velocities[body_i * 2 + 1] += accelerations[body_i * 2 + 1] * DELTA_T;

        positions[body_i * 2 + 0] += velocities[body_i * 2 + 0] * DELTA_T;
        positions[body_i * 2 + 1] += velocities[body_i * 2 + 1] * DELTA_T;
    }
}
```

To update the positions, we first need to update the accelerations and velocities. Following the formulas of physics, this part of the computation is simple and straightforward. All these arrays are stored in global memory, and since each thread accesses unique locations, storing them in shared memory would not be beneficial.

Regarding coalescence, as threads process bodies stored sequentially in the arrays, global memory access is coalesced.

Transferring Data and Deallocating GPU Memory

After updating the body positions, the updated data needs to be transferred to host memory to create the new quadtree in the next iteration:

```
for (int step = 0; step < N_SIMULATIONS; ++step) {
    ...

    // needed for next iteration's tree creation on CPU
    cudaMemcpy(positions.data(), positions_d,
               N_BODIES * N_DIM * sizeof(double), cudaMemcpyDeviceToHost);
}

// deallocation of GPU memory
cudaFree(masses_d);
cudaFree(positions_d);
cudaFree(velocities_d);
cudaFree(accelerations_d);
cudaFree(forces_d);
cudaFree(quadtree_d);
```

After finishing all the steps of the simulation, GPU memory is deallocated.

Results

As we already mentioned, this implementation handles two scaling approaches:

1. A constant number of bodies while increasing the number of threads.
2. Both bodies and threads increase with a one-to-one mapping of threads to bodies.

To provide a more detailed analysis, we compared three different configurations of optimization techniques. Additionally, since a computationally intensive part of the implementation—the quadtree creation—was left out on the CPU, we separated measured times for the entire computation (CPU and GPU) and the GPU-only computation.

Scaling Approach with Constant Number of Bodies with Increasing Number of Threads

For this approach, we used a constant number of bodies equal to **40,000**. The number of threads increases from **1** to **40,000** in powers of **2**. We evaluated three different configurations:

- No shared memory is used, and all quadtree accesses in the force computation kernel are performed in global memory.
- Shared memory is used, but the transfer of the quadtree from global to shared memory is not optimized.
- Shared memory is used with optimized transfer by removing bank conflicts and using coalesced access in global memory during transfer.

For each number of threads, the simulation was executed five times, and the median runtime was selected to minimize the influence of outliers caused by factors such as caching effects, GPU warm-up, and background system processes.

Performance Metrics

The table below presents the results in terms of the best median runtime, best speedup, and best efficiency across all three configurations and two measurement categories. GPU Parallel Timings measure only the time spent in the GPU kernels, while Total Timings include additional overheads such as quadtree construction on the CPU, data transfers, and kernel launches.

Metric	Global Only	Global + Shared (Non-optimized)	Global + Shared (Optimized)
Constant problem size [40,000], increasing processors [1-40,000] - GPU only			
Best Median Runtime (ms, n_threads)	**329.067** (40,000)	**103.629** (40,000)	**64.999** (2,048)
Max Speedup (n_threads)	**797.38** (40,000)	**2,984.15** (40,000)	**921.42** (2,048)
Max Efficiency (n_threads)	**0.92** (2)	**1** (2, 4, 8, 32)	**1** (2, 3, 4)
Constant problem size [40,000], increasing processors [1-40,000] - Total			
Best Median Runtime (ms, n_threads)	**1,331.000** (40,000)	**1,153.000** (40,000)	**1,121.000** (4,096)
Max Speedup (n_threads)	**197.93** (40,000)	**269.19** (40,000)	**54.73** (16,384)
Max Efficiency (n_threads)	**0.91** (2)	**1** (2)	**0.98** (2)

Table 1: Performance results for constant problem size (40,000 bodies) with varying number of threads and configurations.

Analysis of GPU-Only Measurements

We first focus on the GPU-only measurements. The results of the median runtimes are expectedly improving with more advanced memory configurations. For the usage of global memory only, the best median runtime was around **330 ms**, which was achieved in the last scaling step when the number of threads equaled the number of bodies (**40,000**).

For non-optimized shared memory, the runtime improved significantly, with the best around **103 ms**. In the case of optimized shared memory, although the improvement was not as drastic, it was still significant compared to the non-optimized configuration. These values confirm that the addition of shared memory was beneficial, as expected, since each thread traverses the quadtree during force computations and accesses multiple nodes.

One noticeable difference is that for the optimized shared memory, the best runtime was recorded with **2,048 threads**, rather than **40,000** as in the previous two configurations. This is likely due to the combination of:

- More efficient parallel access by removing bank conflicts,
- Reduced global memory latency by introducing coalesced access,
- Less resource contention by having a smaller number of threads.

Speedup Analysis

The speedup results reflect the same effect in terms of the number of threads. However, the speedup is much higher in the non-optimized shared memory configuration. Despite this, the runtime is slower. As speedup is relative to the runtime of a single thread, $T(1)$, this indicates that the earlier iterations of the non-optimized shared memory configuration were the slowest.

This can be attributed to the fact that, compared to the global-only approach, at lower thread counts, a few threads first need to transfer the entire quadtree from global to shared memory, introducing an overhead. In contrast, the global-only configuration allows threads to start computing forces immediately. Furthermore, compared to the optimized shared memory configuration, it is slower due to the non-optimized transfer process.

Efficiency Analysis

The efficiency column shows the expected highest values at the smallest thread counts, with the longest consistency for the shared non-optimized configuration. Increasing the number of threads initially directly helps overcome the drawbacks mentioned earlier.

Analysis of Total (CPU and GPU) Computation

These were the results from the GPU computation measurements. Looking at the results of the total computation (CPU and GPU), we observe the same trend in terms of best median runtimes; however, the ratio between them differs.

The CPU execution overhead remains essentially the same, regardless of the number of GPU threads launched. As the number of GPU threads increases, the fixed CPU overhead becomes a larger fraction of the total runtime, reducing the overall speedup.

Interestingly, the speedup of the global-only approach is higher than the optimized shared memory approach in total computation. This is because longer initial GPU runtimes result in higher relative speedups.

Visualization of Speedups

The plotting visualizations below illustrate how the speedups behave over different numbers of threads used. The data is represented on a logarithmic scale for easier observation of linear slopes.

The dashed red curve represents the ideal linear speedup, defined as $\boldsymbol{S(P) = P}$. The green area below this curve represents the typical behavior where $\boldsymbol{1 < S(P) < P}$, where most parallel algorithms fall. The yellow area represents super-linear speedup with $\boldsymbol{S(P) > P}$, which can occur in some cases but not consistently. Additionally, the red area, where $\boldsymbol{S(P) < 1}$, indicates erroneous parallelism, suggesting that the algorithm should be reconsidered.

Parallel Speedups

Global Memory-Only Approach The global-only approach achieves speedups in the range of **700** to **800x**, with the highest observed speedup of **797x** using **40,000** threads. Since the kernel relies entirely on global memory accesses, its performance is constrained by global memory bandwidth and latency. Although performance scales well from 1 thread to thousands, eventually the memory system becomes saturated, preventing further improvements with additional threads.

Non-Optimized Shared Memory Approach In the non-optimized shared memory approach, single-thread execution has a higher baseline runtime of approximately **310 ms**, compared to **260 ms** in the global-only approach. This increase is due to the additional time required to transfer the quadtree into shared memory, which cannot be hidden by other threads at low thread counts.

As the number of threads increases, the overhead is better hidden, resulting in much larger speedups. A significant jump is observed when scaling from **32,768** to **40,000** threads, possibly due to achieving better occupancy and more effective GPU utilization.

Optimized Shared Memory Approach The optimized shared memory approach demonstrates much faster single-thread execution, with runtimes around **60 ms**, highlighting the advantages of the optimizations applied. At higher thread counts, speedups reach approximately **850x**, which, although smaller than the non-optimized approach, results in the smallest execution times among all configurations.

This configuration provides the best raw performance, even if the relative speedup is not the largest.

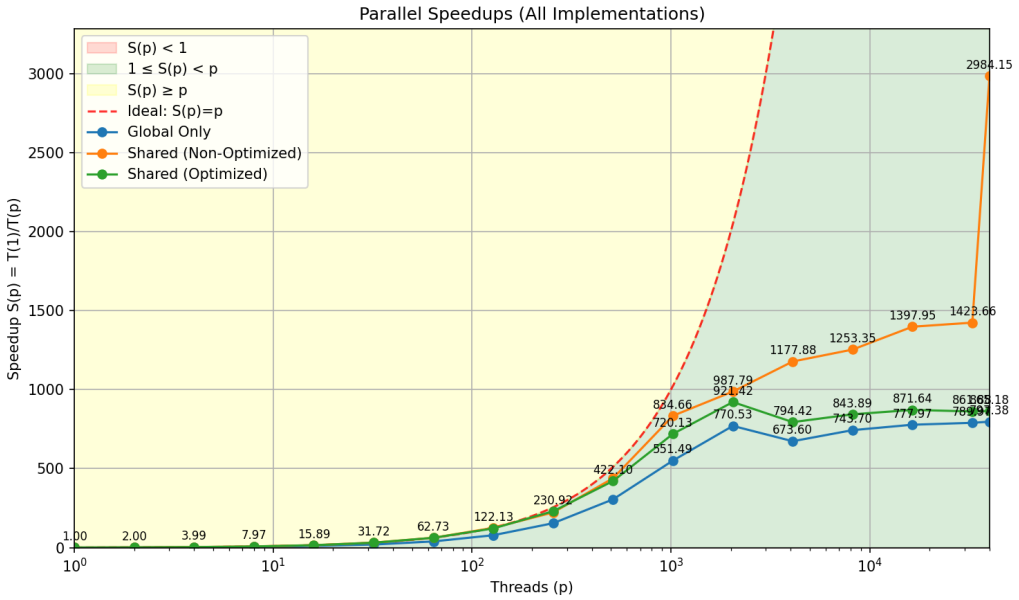

Figure 4: GPU only computation: Speedup achieved per number of threads

Total Speedups

The global-only approach achieves the highest speedup of approximately **190x** for the highest number of threads used. As the GPU kernel portion of the computation becomes smaller, the fixed CPU overhead starts to dominate, resulting in overall smaller speedups across all configurations.

Global Memory-Only Approach In the global-only approach, speedups are limited due to the constant CPU overhead, which becomes more significant as the number of GPU threads increases. Despite this limitation, the maximum observed speedup reaches around **190x**.

Non-Optimized Shared Memory Approach The non-optimized shared memory approach still achieves substantial speedup but is less dramatic compared to the GPU-only measurements. This is attributed to the aforementioned CPU overhead effect. However, it still outperforms the global-only approach due to the advantages provided by shared memory usage.

Optimized Shared Memory Approach The optimized shared memory approach reaches a peak speedup of approximately **50x**. Although its speedups are the lowest among the three configurations, it provides the best raw performance. This is due to its significantly lower single-thread runtime, which ultimately results in the fastest execution times despite the lower relative speedup.

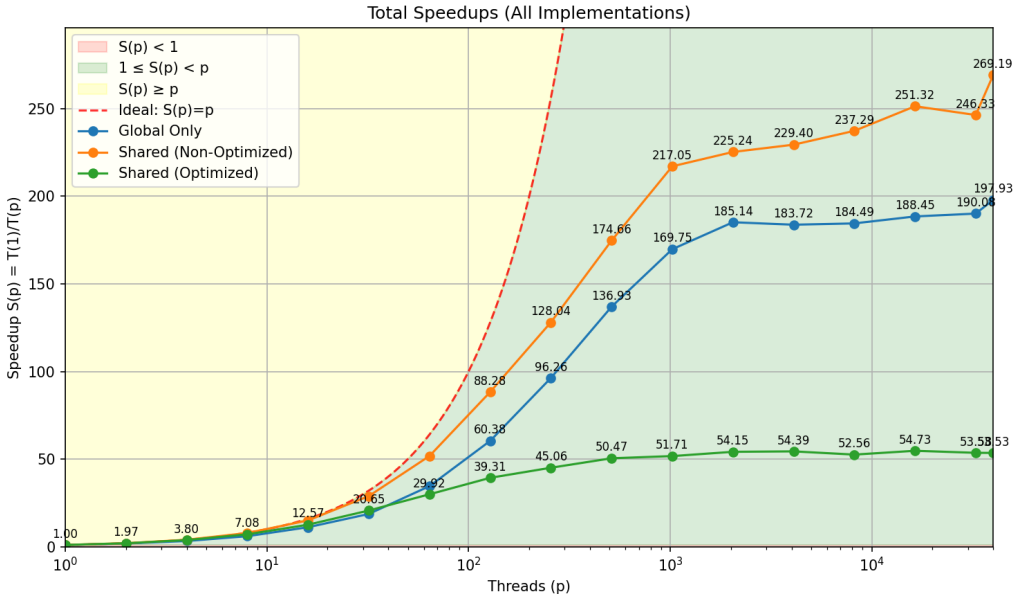

Figure 5: Total computation: Speedup achieved per number of threads

Scaling Approach with Increasing Number of Threads and Bodies

In the second scaling approach, we increase both the number of threads and the number of bodies simultaneously. Our focus here is on comparing execution runtimes across different problem sizes. Ideally, the runtime would remain consistent across all problem sizes.

We began with a starting value of **2**, since a value of **1** would not make sense—having only one body would result in no pairs to compute forces with. The maximum value considered in this approach remains the same as in the first scaling approach, **40,000**.

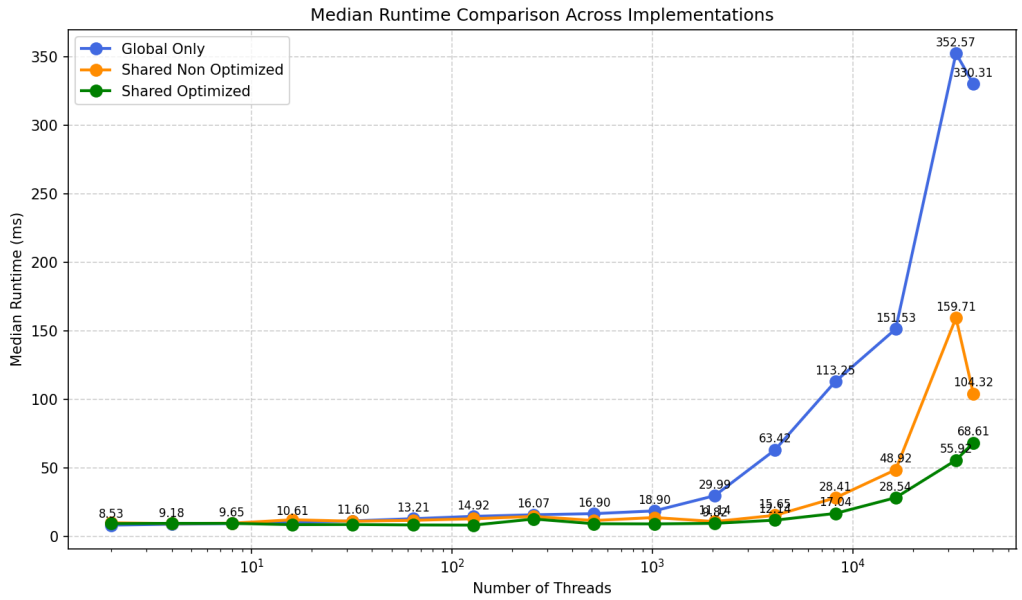

Figure 6: GPU only computation: Runtime per number of bodies/threads

Runtime Analysis for Increasing Problem Sizes

For the global-only approach, runtime gradually increases from around **8.5 ms** to **19 ms** at **1,024** threads. However, a significant jump is observed at the next increasing steps. While the performance might not appear problematic for smaller sizes, as the problem size grows, global memory limitations become a bottleneck, leading to the worst performance among all configurations.

The non-optimized shared approach records a more stable increase in runtime and starts diverging at **4,096** threads, where a large jump is followed

by a significant decrease in runtime, as previously mentioned in the analysis. The overhead of copying data to shared memory, combined with bank conflicts, can cause fluctuations in runtimes.

The optimized shared approach again demonstrates the best raw performance, achieving the smallest runtimes due to applied optimizations. However, noticeable jumps are still present, such as when increasing the size to **32,768**. These occasional fluctuations occur because certain numbers of threads can reduce concurrency by limiting the number of blocks per streaming multiprocessor (SM). In contrast, the next larger number of threads may reorganize resources and improve scheduling efficiency.

Overall, as the problem size increases, the quadtree becomes larger, meaning each kernel call processes more data and makes more memory requests. The fluctuations in runtimes are caused by various factors, including GPU occupancy, block size alignments, shared and global memory usage, and access patterns.

Conclusion

The primary objective of this project was to implement and optimize an N-body simulation using the Barnes-Hut approximation to efficiently compute gravitational interactions among bodies. The exercise successfully achieved its intended purpose by demonstrating significant performance improvements through GPU parallelization and memory optimization techniques. By leveraging shared memory and optimizing memory access patterns, the simulation achieved substantial speedups compared to the naive approach, with the best runtime observed when using 2048 threads. The results highlighted the trade-offs between memory allocation strategies and computational efficiency, showing that while shared memory can significantly reduce global memory latency, improper usage may introduce bottlenecks such as bank conflicts.

Overall, the project provided valuable insights into GPU optimization techniques, including the importance of memory hierarchy, parallel workload distribution, and efficient data structures for spatial decomposition. The experience reinforced the understanding that achieving peak performance requires careful tuning of both hardware resources and algorithmic design. Future improvements could focus on parallelizing the quadtree construction phase and further optimizing memory utilization to enhance overall scalability.