N-BODY SIMULATION USING BARNES HUT AND A SPHERICAL OCTREE

Lea Holter, Orion Junkins, Cyril Moser, Nicola Studer

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The N-Body problem describes the gravitational influence of n celestial bodies on each other. It is traditionally solved using the Barnes-Hut algorithm – an $O(n \log n)$ method that leverages spatial partitioning and aggregate body approximations. This work presents a novel approach to vectorization and hyper-optimization of the original Barnes-Hut algorithm, using a spherical octree as simulation driver. Our method employs a particle-vectorized traversal of the octree and a memory-optimized layout for both tree nodes and particles, achieving speedups of $\geq 50\times$ for small n and $\geq 15\times$ for large n.

1. INTRODUCTION

The N-Body problem arises in various scientific areas, such as astrodynamics [1], machine learning [2], molecular dynamics [3], and fluid dynamics [4]. In general, it describes the motion of n bodies over time, each influencing each other gravitationally or electrostatically.

It is inherent that the simulation for a large amount of particles needs to be computationally feasible to achieve real-time simulations. Whereas the trivial brute force algorithm has a quadratic runtime, the Barnes-Hut algorithm reduces this to $O(n \log n)$ [5] by spatially partitioning the particles with an octree and gathering far away particles as a single mass point, thus enabling efficient force evaluation.

However, getting a fast implementation of the Barnes-Hut algorithm is challenging, as building and traversing the octree with vectorized instructions is non-trivial. Common methods either use a modified, more vectorizable variant of the Barnes-Hut algorithm [6] or port the code to Multicore CPUs [7] or even GPU [8].

Contribution. We propose a modified variant of the Barnes-Hut algorithm, using a spherical octree as the driver of the simulation, which achieves speedups of $\geq 50\times$ for small n and $\geq 15\times$ for large n compared to our baseline. Making use of AVX-512 [9], our modified code has a completely vectorized traversal of the tree, fast integration and coordinate transformations. Using careful arrangement of tree nodes, we get linear access patterns over the data throughout the seemingly random octree traversal.

2. BACKGROUND ON THE ALGORITHM

This section introduces the mathematical and physical foundation of the N-Body problem, as well as the basic idea behind the algorithm used.

Physical Description of the N-Body problem. The N-Body problem can be physically described by the aggregated attraction forces working on each particle i, described by Newton's law of universal gravitation [10]:

$$\mathbf{F}_i = G \sum_{j \neq i} \frac{m_j m_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2^2} \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|_2}, \tag{1}$$

where $\mathbf{x}$ denotes the particle positions, m their mass and $\mathbf{F}$ the aggregated force. Using Newton's second law of motion [11] $\mathbf{F} = m \cdot \frac{\partial^2 \mathbf{x}}{\partial t^2}$ and symplectic Euler integration [12], we can express the updated velocity $\mathbf{v}$ and position $\mathbf{x}$ at time $t+1$ as:

$$\mathbf{v}_i^{(t+1)} = \mathbf{v}_i^{(t)} + \frac{1}{m_i} \mathbf{F}_i \cdot \Delta t, \tag{2}$$

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t)} + \mathbf{v}_i^{(t+1)} \cdot \Delta t, \tag{3}$$

using Δt as the timestep. The construction of eq. (1) imposes a trivial $O(n^2)$ algorithm, which makes it prohibitively expensive for large systems.

To address computational bottlenecks, Barnes and Hut introduced a $O(n \log n)$ algorithm [5], based on spatial decomposition. The domain is recursively divided into a three dimensional octree [13], where each node represents a region of space containing a subset of particles. For distant nodes, the influence of a group of particles can be approximated by a unified mass point. Specifically, for a distant node containing a set of particles $\mathcal{N}$, the total mass M_j and center of mass $\mathbf{C}_j$ are:

$$M_j = \sum_{i \in \mathcal{N}} m_i, \qquad \mathbf{C}_j = \frac{1}{M_j} \sum_{i \in \mathcal{N}} m_i \mathbf{x}_i. \tag{4}$$

The acceleration from particle i to this mass point can be expressed as:

$$a_{i \to j} \approx G M_j \frac{\mathbf{C}_j - \mathbf{x}_i}{\|\mathbf{C}_j - \mathbf{x}_i\|_2^3}. \tag{5}$$

The accuracy of the hierarchical approximation can be controlled via a threshold parameter that determines when a region is sufficiently far and small to apply the approximation. The algorithm is described in algorithm 1.

Algorithm 1 Barnes-Hut Algorithm, single step

```
Require: Particle positions $\mathbf{x}$, velocities $\mathbf{v}$, masses $m$; timestep $\Delta t$
 1: $T \leftarrow$ BUILDOCTREE($\mathbf{x}, m$)
 2: for each particle $i$ do
 3:     $\mathbf{a}_i \leftarrow \mathbf{0}$                         ▷ Initialize acceleration
 4:     $S \leftarrow$ CREATESTACK(GETROOT($T$))
 5:     for each node $j$ in $S$ do
 6:         if ISLEAF($j$) then
 7:             $\mathbf{a}_i \leftarrow$ GETACC($j$)            ▷ Using eq. (1)
 8:         else if ISFARENOUGH(i, j) then
 9:             $\mathbf{a}_i \leftarrow$ GETAPPROXACC($j$)      ▷ Using eq. (5)
10:         else
11:             ADDCHILDREN($S$, $j$)
12:         end if
13:     end for
14:     $\mathbf{v}_i \leftarrow$ INTEGRATEVEL($\mathbf{a}_i, \delta t$)         ▷ Using eq. (2)
15:     $\mathbf{x}_i \leftarrow$ INTEGRATEPOS($\mathbf{v}_i, \delta t$)         ▷ Using eq. (3)
16: end for
Output: Updated particle positions $\mathbf{x}$ and velocities $\mathbf{v}$
```

Spherical Octree Restriction. As per the project description, the task was extended with the restriction of building and maintaining the octree in spherical coordinates. This directly imposes a change of coordinates, as described in [14]. It would be out of scope to rewrite the dynamics of the system (eqs. [1, 2, 3]) in spherical coordinates; hence, we have opted to use coordinate transformations. Therefore, throughout the simulation, the dynamics stay in Cartesian coordinates, whereas the data structures lie in spherical coordinates. Using the spherical octree variant together with algorithm 1, we defined a baseline implementation using a pointer-based "*Arrays of Struct*" (AOS) style.

Cost Measure. Defining the cost measure of the Barnes-Hut algorithm is no trivial task, as even if we fix the number of particles n, the flop count can vary depending on the initial condition, i.e., position, mass, and velocity of the particles. The reason for this is that, depending on the position of the particles, the tree can vary its size. Thus, the performance is not the most accurate metric to describe the influence of the optimizations done in section 3. Therefore, we opt to use cycle counts as our cost measure. As described in [15], instead of plotting just the mean or the median, we also include the maximum and minimum to show the measurement variation.

Flops and other metrics are counted with `perf` [16]. The time distribution of various sections of the running code is visualized and measured with `flamegraph` [17].

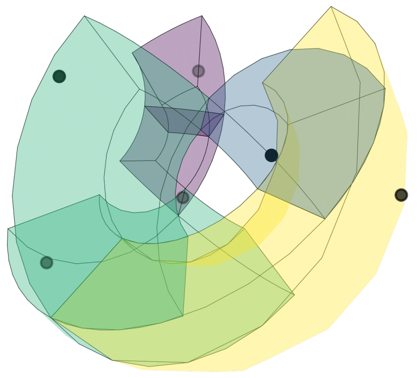

Fig. 1: Graphical visualization of implemented octree.

3. OPTIMIZATIONS PERFORMED

In this section, we introduce various optimizations we have performed from the baseline up, including our various approaches in SIMD (Single Instruction Multiple Data) tree traversal, stack management and precomputations.

Octree. The spherical coordinate octree, visualized in figure 1, is the core data structure of this algorithm. For the basic implementation introduced above, each tree node is stored as a separate struct instance with pointer references to other node instances. This allows a single node's data to fit within a small number of cache lines. While optimal for the baseline algorithm presented above (where nodes are processed individually), this format does not lend itself well to SIMD parallelism. Thus, the first step of our optimization was to transition to a *Struct of Arrays* (SOA) format.

Transitioning to SOA requires contiguous memory allocation. This memory can be allocated statically by precomputing the required space for a fully dense tree. Due to the branching factor of 8, the memory consumption grows exponentially, and the tree depth must be bounded ≤ 8, limiting the simulation fidelity. However, the final tree tends to be sparse, so this allocation is excessive. A better approach is to reallocate as needed dynamically. We allocate space for $\#particles + 16$ nodes, where the addition of 16 avoids power-of-two stride issues. We maintain the same allocated memory for the entire simulation and re-allocate if a particular timestep requires more memory. In practice, this results in a small number of re-allocations (generally ≤ 10) early in the simulation, and rarely any thereafter.

We also optimize two aspects of the tree using SIMD. First, we apply SIMD to parallelize the initialization of the arrays (which must be performed every timestep). Second, we apply SIMD during node subdivision, where values are computed and set for 8 contiguous children at once. As our results and discussion in section 5 will demonstrate, simply transitioning to SOA format does not immediately provide a speedup. However, this restructuring is a critical first step that enables the subsequent optimizations.

AVX Force Integration and Coordinate Transforms. Instead of integrating the force for particles individually, we can make use of the SoA format by temporarily storing the acceleration in corresponding arrays. After the tree traversal is done for all particles, we can AVX over the particles linearly and use eq. (2) and eq. (3) to integrate the positions and velocities using vectorization and FMAs.

According to `flamegraph` [17] estimates, the baseline implementation spends more than 60% of cycles doing the conversion from spherical to Cartesian coordinates, which needs expensive trigonometric functions. This happens because the conversion is called each time when either calculating a part of eq. (1) or eq. (5). Considering that a node is likely to be traversed multiple times, we are recomputing the Cartesian position of nodes multiple times. Therefore, precomputing all Cartesian node positions and storing them in a contiguous array is reasonable, as this is trivially vectorizable as well. As not all trigonometric functions are implemented for AVX-512, we have opted to include the implementations of SLEEF [18].

The same approach can also be applied to the particles; we can maintain a Cartesian representation of a particle's position in parallel to the spherical one across timesteps. Getting rid of the conversion from Cartesian to spherical is impossible, as building the tree is done in spherical coordinates.

Lookup Tables. While precomputing Cartesian node positions has decreased the number of cycles required for switching coordinate systems, the actual transformations are still very expensive. In theory, each conversion consists of 4 $\sin()$ and $\cos()$ calls in total. Although they can be merged using the `sincos_fma` instruction, they are still very slow [19]. Considering that our algorithm is an approximation, we can circumvent this issue by using lookup tables for trigonometric values. More specifically, we store a fixed number k number of $\sin(x)$ values, where $x \in [0, \frac{\pi}{2}]$. This works because of the symmetry of $\sin(x)$, and the identity $\sin(x + \pi/2) = \cos(x)$. For general lookup values $x \in [0, 2\pi]$ we use the following mapping:

$$\texttt{id} = \left(\frac{x}{2\pi}\right)(4k-1), \tag{6}$$

$$\text{lookup}(\texttt{id}) = \begin{cases} \text{table}[\texttt{id}] & \texttt{id} \in [0, k[\\ \text{table}[2k - \texttt{id} - 1] & \texttt{id} \in [k, 2k[\\ -\text{table}[\texttt{id} - 2k] & \texttt{id} \in [2k, 3k[\\ -\text{table}[4k - \texttt{id} - 1] & \texttt{id} \in [3k, 4k[\end{cases} \tag{7}$$

First, we define `id`, giving an index into the lookup table relative to 2π. This `id` imposes in which quadrant we are in to get the final index in the table, and also the sign of the resulting value.

Finally, to further reduce runtime, we utilize vectorization to perform multiple lookups in parallel. This is made possible through the previous optimization, where we precompute all the conversions before the actual loop starts, allowing for easy batching.

AVX Traversal. With the simple vectorization already performed in the previous optimizations, the remaining scalar part was the tree traversal. Under the assumption that close particles traverse the tree in a similar way, i.e., most of them take the Barnes-Hut approximation at the same time, we vectorized the traversal over a group of 8 or 16 particles, depending on datatypes. To ensure equivalence to our baseline (up to floating point precision), we need to deal with traversal situations where some of the particles continue traversing the child nodes while some reach the Barnes-Hut threshold and skip them. We solved this by continuing traversal with child nodes so long as any of the vectorized particles require it. We keep a mask for each depth layer of the tree to skip over those nodes where the Barnes-Hut threshold is already reached. These depth masks get reset with the mask from the upper layer when starting the processing of a new node, as even if a particles skips a certain subtree from that node on out, we cannot guarantee such a thing for other subtrees at the same depth, as seen in figure 2.

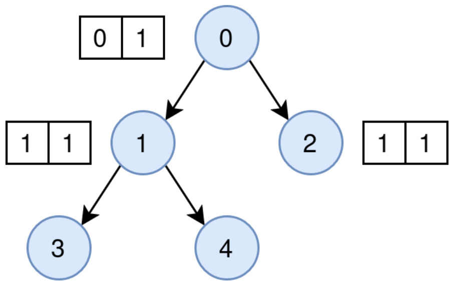

Fig. 2: Depth masks for 2 particles, assuming the second particle has already been calculated. For computation of node 2, the mask is reset to the mask of node 0. Both particles take the approximation, so nodes 3 and 4 are skipped.

It is additionally worth noting that our traversal approach can be adapted to provide even higher accuracy without changing runtime. All ”wasted compute” occurring in SIMD lanes corresponding to finished particles could be retained to increase accuracy, resulting in an algorithm that is *lower bounded* by the classical Barnes-Hut approximation, but gives higher accuracy wherever the compute is already occurring. To not deviate from the accuracy of the base implementation, we use the depth masks.

Particle Sorting. The aforementioned tree traversal vectorization only offers a speedup if the particles being traversed in parallel take similar traversal paths through the tree. In practice, spatially adjacent particles tend to follow similar traversal paths. To exploit this, we sort particles to improve spatial locality and maximize SIMD utilization.

We sort using two approaches. First, we do a basic sorting that considers the x component first, then y if the x component is equal, then z if the y component is also equal. In practice, this effectively sorts by x position and is not particularly optimal, but it provides a good starting point. Next, we implement a Morton code-based approach [20]. Cartesian positions are normalized to the range $[0, 1]$, then scaled to integers in the range $[0, 1023]$, yielding 10-bit values per axis. These values are interleaved bitwise - first the most significant bit of x, then y, then z, etc. - to produce a single 30-bit Morton code. Particles are sorted according to these 30-bit representations. Because particle movement between timesteps is typically small, and perfect sorting is unnecessary, we only perform this sorting every 500 timesteps.

Micro-optimizations. Despite removing the expensive sin and cos calls, we still have very expensive calculations in the innermost loop. Consider the code in listing 1.

```
1  diff <- cartesian_node - cartesian_p
2  if node.leaf and not node.com = p.pos:
3      # node is a leaf that does
4      # not store this particle
5      r <- sqrt(diff.SquaredNorm())
6      dv <- diff * G * p.mass / (r*r*r)
7      acc <- acc + dv
8  else:
9      # internal node or node
10     # storing current particle
11     r <- sqrt(diff.SquaredNorm())
12     ratio <- get_size(node) / r
13     if ratio < THRESHOLD:
14         # take the BH approximation
15         dv <- diff * G * p.mass / (r*r*r)
16         acc <- acc + dv
```

Listing 1: The compute heavy part of the innermost loop

We have square roots (lines 5 and 11) and divisions (lines 6, 12 and 15). Specifically, the *VSQRTPD* instruction for computing the square root has a latency of 21 cycles and a reciprocal throughput of 16 cycles. Division instructions *VDIVPD* have a latency of 13 cycles and a reciprocal throughput of 9. The very high latency is a larger problem than the high reciprocal throughput, as subsequent calculations all immediately depend on the results of these instructions. But the crucial observation here is that we only require $\frac{1}{r}$ and not r itself. This allows us to utilize the much faster *RSQRT14PD* instruction, which calculates the inverse square root with a latency of 2. So instead of directly calculating r as before, we now have $\frac{1}{r}$, and we can simply multiply this value instead of having to divide it. This yields the improved code shown in listing 2.

Precomputation of Node Sizes. In the original Barnes-Hut, the octree is not in spherical coordinates; hence, getting the size of a node is a simple lookup of the length of a side of the current octree cell. This is not the case for our spherical octree. Here we calculate the maximal arclength at the middle of the cell (so either in the theta or psi direction). As this computation must be performed for each node and involves a long dependency chain, precomputing these node sizes is another straightforward optimization we can implement. This also enables us to vectorize the computation, which would not be possible otherwise.

```
else:
    # internal node or node
    # storing current particle
    r_inv <- rsqrt14(diff.SquaredNorm())
    ratio <- get_size(node) * r_inv
    if ratio < THRESHOLD:
        # take the BH approximation
        r_inv_cubed <- r_inv * r_inv * r_inv
        dv <- diff * G * p.mass  * r_inv_cubed
        acc <- acc + dv
```

Listing 2: Optimized else block using RSQRT14PD

DFS-Sorting. For each particle, we traverse the tree in the same depth-first search (DFS) order. The only differences between different traversals are when we decide to skip subtrees. This optimization exploits that. After building the tree, it sorts all the nodes according to this DFS traversal order. As we are now using SOA, this means that we have to sort all arrays associated with the nodes. For illustration purposes, we view the nodes as a single memory location here, though it works the same with SOA.

This sorting of the nodes has proven to be rather cheap, taking up at most 5% of our cycles. But the benefits are very noticeable. First off, this allows for much better spatial locality for caches, as we linearly traverse the different arrays corresponding to the tree nodes. Previously we had no such guarantees. The only guarantee we had was that all the children of a node are consecutive in memory. This makes building the tree a lot easier, but is not very useful for cache performance, due to DFS. Specifically, since DFS goes into depth first, it might take a while until it comes back to the next sibling at the current depth again, as it had to process the entire subtree first. Until then, some cache lines corresponding to certain node values may have already been evicted. Considering that we have 13 arrays storing floating point values alone, with each entry corresponding to one node, there is bound to be some conflict following from all the random accesses. With the new DFS ordering such conflicts are less likely due to our linear stride.

Additionally, this ordering allows us to completely remove the stack used in the main Barnes-Hut (BH) loop. This works because if there are no BH-approximations that are taken, we can simply traverse the node array one by one. And if there is a BH-approximation, we can simply jump ahead by the size of the subtree, which we can easily store per node. Assuming we take the BH-approximation at node 1, we get what is visualized in figure 3. As we can see, jumping ahead by the size of the subtree (here 2) is equivalent to not pushing the children of that subtree on the stack.

In both cases we continue with the DFS at the correct node. This also allows us to remove any nodes with mass 0, i.e. that contain no particles. So, the tree is much more densely stored, and additionally, no computation is wasted with our vectorization code over nodes (as in the precomputation of node sizes and coordinate systems transformations).

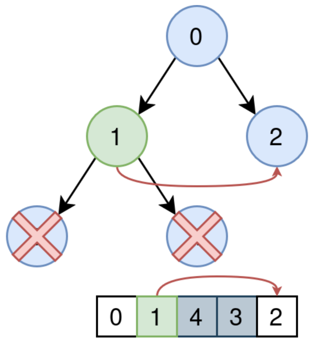

Fig. 3: Skipping ahead in a DFS-Sorted Tree

4. COMPETING OPTIMIZATIONS

In this section, we describe competing optimizations excluded from the final version, because they were outperformed by the approaches described above.

Multi-Stack AVX traversal. For vectorized tree traversal, we explored independently tracking tree traversals for each particle of the current group and loading the appropriate nodes from the tree using a series of *gather* instructions. This allows us to fully utilize the vectorized computations and reduce the flopcount overhead compared to the depth-mask approach detailed above. However, performance evaluation showed that due to the many gather instructions and resulting chaotic memory accesses, the overall runtime was much worse, resulting in an overall slowdown instead of speedup for this implementation.

Vectorized tree traversal. Another alternative for vectorizing tree traversal was to parallelize across siblings for a given subtree. This is relatively straightforward, as (prior to introducing DFS-sorting) all 8 children nodes are stored contiguously in memory. However, during evaluation, we found that this approach was severely limited by the relatively sparse nature of the tree, leading to many unnecessary calculations performed for empty leaf nodes. For this reason, the optimization performed worse compared to our particle-based vectorization. Nevertheless, this version may remain interesting for very large n, as it may display preferential scaling properties as the tree fills up further.

Stack unrolling. Before implementing the DFS-Sorting, we also explored eliminating the stack using a different approach. By implementing the traversal using a recursive preprocessor macro, we get an iterative code that is unrolled across the traversal depth. This replaces the stack-based structure with individual iteration counters and turns the stack-based loop into a series of nested *for* loops, each with 8 iterations for each of the 8 tree children. For this optimization, we also experimented with the `-fno-unroll-loops` flag to avoid unrolling the innermost loops, since they typically get skipped anyway. This gave a minor speedup, but was dwarfed by the speedup of the DFS-Sorting approach. This indicates that the primary benefit of DFS-Sorting is not eliminating the stack, but rather improving cache locality.

5. EXPERIMENTAL RESULTS

This section describes our experimental setup, our results, and discussions exploring those results.

Experimental setup. We evaluate algorithm variants using datasets of randomly generated positions with count $n \in \{128, 256, 512, 1024, 2512, 3024, 4048, 6096, 10192\}$. We use powers of 2 for the small *n*. For $n \geq 1024$ we use powers of 2 plus 2000 to avoid cache aliasing effects caused by power-of-two stride patterns. For faster variants, we evaluate larger particle counts $n \in \{18384, 34768\}$.

For smaller n trials, all experiments use a fixed max tree depth of 8. This relatively low value is required by the statically allocated octree variant, which cannot support deeper trees. For larger n trials, all experiments use a fixed max tree depth of 32, giving a much higher fidelity simulation.

For each *n*, we generate 6 datasets of particles distributed across a fixed bounding box. Each algorithm is run for 500 timesteps, five times independently against each dataset. We take the mean over these five runs. Of these means, we report the minimum, median, and maximum across all 6 datasets for each algorithm for each n. All experiments use a threshold value of 1.0, and a full sort frequency of 500 (i.e., just one initial sort). We use an AMD Ryzen 9 7950X and compile with GCC 14 using the following flags:

```
-O3 -march=znver4 -mtune=znver4 -flto
-fno-omit-frame-pointer -static
```

We also experimented with `-ffast-math`. We found that it gave a minor improvement in less optimized variants by enabling (among others) auto-vectorization by the compiler. However, it had no measurable impact on our manually vectorized versions, so we exclude it.

Results. First, we examine the performance of our baseline (”Naive Algorithm + Pointer-Based Octree”) in comparison to our initial optimizations. Specifically, we consider the naive algorithm with the static and dynamic octrees (”Static Alloc SOA Octree” and ”Dynamic Alloc SOA Octree” respectively), our initial AVX optimizations (”AVX Conversion, AVX Integration and Lookup tables”), and our initial AVX Traversal (”AVX Traversal”). Figure 4 shows the runtime of these variants.

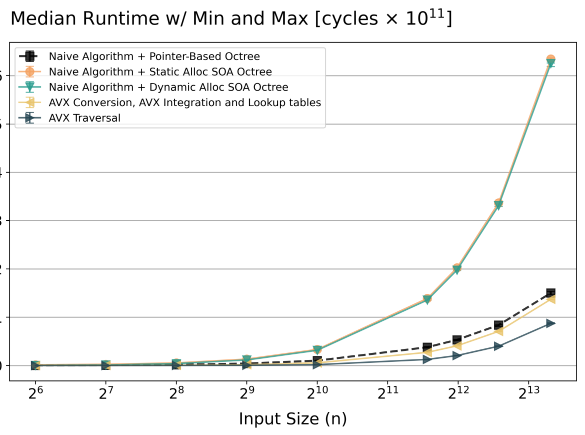

Fig. 4: Runtime of initial optimizations and baseline

Switching the octree to the SOA without further optimization format hurts performance. This is due to a misalignment between the traversal approach and the underlying data structure. The naive algorithm processes a single particle at a time. With the naive octree, an entire node can fit in a few cache lines. However, when using the SOA format, each member must be read from a distinct array, resulting in significantly less efficient memory accesses. Once we adapt parts of our algorithm to leverage the benefits of the SOA format (namely, utilizing AVX for integration and conversion with lookup tables), we outperform the baseline. Our AVX traversal further reduces runtime.

Second, we explore the performance of additional optimizations applied on top of the "AVX Traversal". We introduce the following optimizations, each building on the previous: "Precomputation of Node Size", "Common Subexpression Elimination and Reciprocal Square Roots", "Basic Sorting", "Morton Sorting", "Double to Float", and "DFS Sort". Figure 5 shows the runtime of these variants, and figure 6 shows the speedup. Note that we transition to a larger max depth of 32 and larger n values up to 34768.

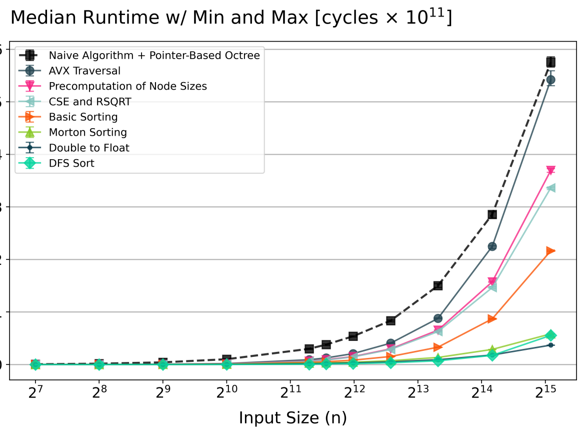

Fig. 5: Runtime of advanced optimizations against baseline

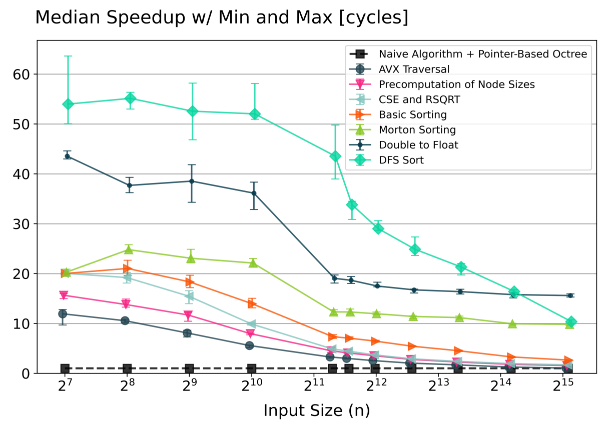

Fig. 6: Speedup of advanced optimizations against baseline

These plots demonstrate the effectiveness of our traversal, as well as its dependence on proper sorting. The naive AVX traversal, even without sorting, achieves $\geq 10\times$ speedup. However, speedup quickly drops off as n grows. While the micro-optimizations increase this speedup across all n, they do not prevent the drop-off as n increases. Adding basic sorting slows this dropoff considerably, but the speedup still degrades as n grows large. Morton sorting, however, yields far more stable speedups for large n. To understand this, we perform an additional experiment. We log the depth mask after processing each item in the stack and compute the average fullness of the mask. The more '0' entries in a depth mask we have, the greater our parallelism and thus the greater our speedup. Figure 7 shows these results.

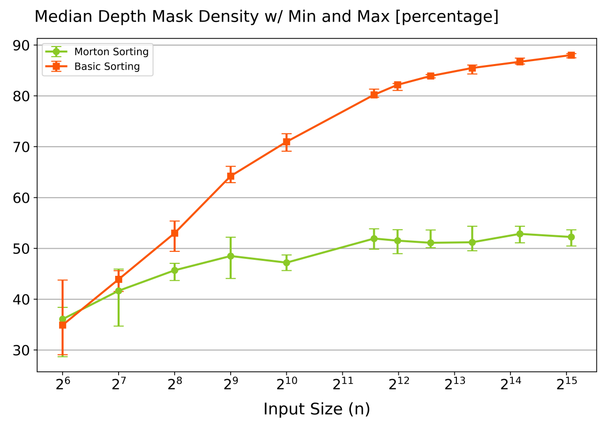

Fig. 7: Mask Density for Basic and Morton sorting

Using our simple sorting, the depth mask approaches 90% fullness for large n (e.g. up to 90% of our lanes were not doing useful computation). Morton sorting, on the other hand, stabilizes at approximately 50% fullness. This means that we are achieving approximately $8\times$ parallelism on average (50% of the $16\times$ possible when using floats).

The results in 6 also demonstrate both the efficacy and a limitation of our DFS approach. For moderate $n \leq 2^{14}$, the DFS ordering yields a large speedup. However, for $n \geq 2^{15}$, DFS sorting is no longer beneficial, and our simpler stack-based "Double to Float" variant performs best. This can be understood through a closer inspection of performance counters. The DFS ordering produces considerably more TLB misses for large n. At $n = 34768$, across the 6 datasets, the "DFS Sort" variant yields a median of $2.2\times$ more cache misses than the "Double to Float" variant (min of $1.3\times$, max of $12.9\times$). Consider how both approaches handle 8 children of a given node, which all meet the approximation threshold. With the stack-based ordering, these 8 children are contiguous in memory, with their subtrees stored in some other, potentially far away, block of memory. Because we are taking the approximation, the stack-based algorithm loads a single page containing these 8 children, evaluates them, and moves on without exploring their subtrees. With the DFS ordering, each child's subtree will be stored immediately following it in memory. If these subtrees are large, the 8 children may live far apart in memory, requiring access to up to 8 unique pages, despite not actually needing the data in between. This access pattern provides the advantages described in section 3, but it comes at the expense of accessing many more pages.

For smaller n, this tradeoff is worth it, as the additional page accesses only occur for the first particle, and TLB misses are relatively rare for the rest of the simulation. But, once n grows large enough that the tree data structure uses up all available pages, then TLB misses become frequent as each particle will overwrite a large portion of the TLB cache, forcing the next particle to refetch all data from scratch. At this point, the stack traversal becomes more optimal as each particle hits fewer total pages; sequential particles are likely to use similar subsets, enabling higher TLB hit rates.

The results in 6 also show a drop in performance from maxing out the 512 KiB L2 cache. The threshold cannot be exactly computed as it depends on the distribution of the particles. However, performance counters consistency show an increase in L2 misses between $n = 2^9$ and $n = 2^{10}$.

For the sake of completeness, we also provide a performance plot of our "DFS Sort" variant in figure 8. For our machine, the peak theoretical single-core performance for 32-bit floats is 64 FLOPs per cycle (2 FMA units, 16-wide SIMD). Our performance remains in the range of 12.5–17.5 FLOPs/cycle across all n. While this is far below the theoretical maximum, several factors limit attainable performance. First, many operations cannot be expressed as FMAs, reducing peak throughput by up to 2× in those code regions. Second, parts of the algorithm are memory-bandwidth bound. For large n, frequent L2/L3 accesses are required, and despite optimization efforts, ideal cache coherence cannot always be maintained. Third, some sections still exhibit dependency chains, leading to latency-bound execution. Note that performance is a misleading metric for this algorithm; the results of many flops are 0'd out by the depth mask.

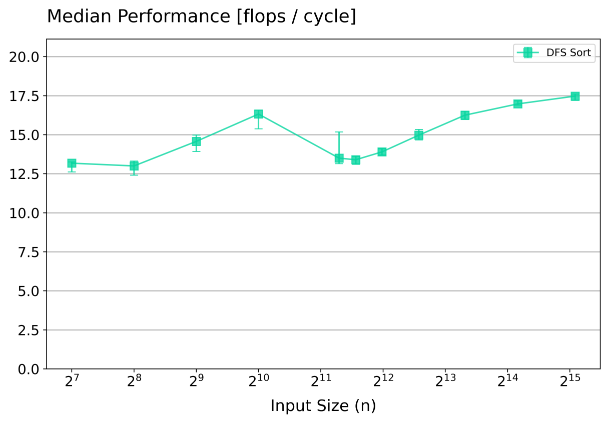

Fig. 8: Performance of the best performing variant

6. CONCLUSIONS

We introduce a series of optimizations for the Barnes-Hut approximation of the N-body problem using spherical coordinates. Our AVX traversal, in combination with numerous smaller optimizations, yields a $\geq 15\times$ speedup for arbitrary n. Additionally, we introduce a DFS-based node ordering that achieves a $\geq 50\times$ speedup for moderate n. However, this DFS-based approach depends on the octree data fitting within the TLB cache, resulting in a considerable loss of efficiency when n becomes large enough to cause widespread TLB misses. Thus, the most optimal algorithm depends on the size of the input dataset; moderately sized simulations benefit from the DFS Sorting optimization, while very large systems are better off without this final optimization. Future work could explore decomposing the tree traversal of the DFS Sorted nodes to leverage the benefits of the DFS ordering without excessive causing TLB misses.

As with the original Barnes-Hut approach, our optimized variants can only approximate the n^2 algorithm result. However, by tuning the maximum tree depth, the lookup table fidelity, and the BH-threshold value the quality of the approximation can be tuned arbitrarily in exchange for longer runtimes. Furthermore, our algorithm could reach higher accuracy than Barnes-Hut without additional computation. Computations being performed in "empty" SIMD lanes could be leveraged to refine accelerations for particles beyond the point that the standard BH algorithm would approximate. But this would deviate from the true Barnes-Hut algorithm.

Finally, it is worth noting that our approach could be applied to other tree-traversal-based problems. Many of our core optimizations are generic with potential application to other algorithms and domains.

7. CONTRIBUTIONS OF TEAM MEMBERS (MANDATORY)

This section briefly describes the contributions done from each team member.

Lea. Implementing alternative tree traversal approaches, vectorized traversal over node children instead of particles. Using multiple stack for vectorized particles. Implemented an unrolling stack variant to improve super scalar execution. Compiler flag experimentation. Performance measurement framework (incl. flamegraphs).

Orion. Implemented baseline data structures and Cartesian Octree. Implemented numerous optimizations on the Spherical Octree: Transition to SOA format, memory management improvements, AVX application to subdivision and initialization. Assisted Nicola in the ideation of depth mask maintenance for the AVX traversal algorithm. Implemented particle sorting algorithms to improve AVX traversal efficiency. Investigated drop in performance in DFS sorted variant due to increased TLB misses.

Cyril. Implemented spherical octree baseline based on Cartesian octree. Designed cost metric with Nicola for the node sizes. Implemented lookup tables and their vectorization. Explored various versions there. Implemented Micro-Optimizations. Did other smaller improvements to various things, like the stack. Implemented DFS-Sorting.

Nicola. Implemented baseline algorithm using data structures from Orion and Cyril. Implemented vectorized integration and Cartesian to spherical coordinate transformation. Blackboard brainstorming about particle vectorized tree traversal with the group, solo implementation with multi-layer depth masks. Python interface to visualize.

8. REFERENCES

[1] Richard H Battin, *An introduction to the mathematics and methods of astrodynamics*, Aiaa, 1999.

[2] Parikshit Ram, Dongryeol Lee, William March, and Alexander Gray, “Linear-time algorithms for pairwise statistical problems,” *Advances in Neural Information Processing Systems*, vol. 22, 2009.

[3] John A Board Jr, Christopher W Humphres, Christophe G Lambert, William T Rankin, and Abdulnour Y Toukmaji, “Ewald and multipole methods for periodic n-body problems,” in *Computational Molecular Dynamics: Challenges, Methods, Ideas: Proceedings of the 2nd International Symposium on Algorithms for Macromolecular Modelling, Berlin, May 21–24, 1997*. Springer, 1999, pp. 459–471.

[4] Orlando Ayala, Wojciech W Grabowski, and Lian-Ping Wang, “A hybrid approach for simulating turbulent collisions of hydrodynamically-interacting particles,” *Journal of Computational Physics*, vol. 225, no. 1, pp. 51–73, 2007.

[5] Joshua Barnes and Piet Hut, “A hierarchical o(n log n) force-calculation algorithm,” vol. 324, no. 6096, pp. 446–449, Publisher: Nature Publishing Group.

[6] Joshua Barnes, “A modified tree code: don’t laugh; it runs,” *Journal of Computational Physics*, vol. 87, no. 1, pp. 161–170, 1990.

[7] Junchao Zhang, Babak Behzad, and Marc Snir, “Design of a multithreaded barnes-hut algorithm for multicore clusters,” *IEEE Transactions on parallel and distributed systems*, vol. 26, no. 7, pp. 1861–1873, 2014.

[8] Abhishek Madan, Nicholas Sharp, Francis Williams, Ken Museth, and David IW Levin, “Stochastic barnes-hut approximation for fast summation on the gpu,” *arXiv preprint arXiv:2506.02219*, 2025.

[9] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2: Instruction Set Reference*, 2024, Available at `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[10] Fritz Rohrlich, *From paradox to reality: Our basic concepts of the physical world*, Cambridge University Press, 1989.

[11] Jerry B Marion, *Classical dynamics of particles and systems*, Academic Press, 2013.

[12] Ernst Harier, Christian Lubich, and Gerhard Wanner, “Geometric numerical integration,” *Structure-Preserving Algorithms for Ordinary*, 2000.

[13] Donald JR Meagher, *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*, Electrical and Systems Engineering Department Rensseiaer Polytechnic ..., 1980.

[14] Matt Pharr, Wenzel Jakob, and Greg Humphreys, *Physically based rendering: From theory to implementation*, MIT Press, 2023.

[15] Torsten Hoefler and Roberto Belli, “Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results,” in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2015, pp. 1–12.

[16] "Linux perf tool," `https://perf.wiki.kernel.org/`, Accessed: 2025-06-14.

[17] Brendan Gregg, "Flamegraphs," `http://www.brendangregg.com/flamegraphs.html`, 2016, Accessed: 2025-06-14.

[18] Naoki Shibata and Francesco Petrogalli, "Sleef: A portable vectorized library of c standard mathematical functions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1316–1327, 2019.

[19] Intel Corporation, "sincos_pd - intel® intrinsics guide," `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=sincos&ig_expand=6094`, 2025, Accessed: 2025-06-14.

[20] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," Technical Report IBM Report, IBM, 1966.