BOOST-HUT - MICROARCHITECTURALLY OPTIMIZED N-BODY SIMULATION

Emanuel Mairoll, Niels Pressel, Daniel Baciu, Berndt Uhlig

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We present a highly optimized single-threaded implementation of the Barnes-Hut N-body simulation algorithm, explicitly designed to utilize modern microarchitectural CPU features. Our approach tackles the computational bottlenecks of this algorithm, such as expensive trigonometric operations, overhead of recursive function calls, and inefficient memory access patterns. To address these challenges, we utilize a optimized data structure (FlatTree), iterative tree traversal, spatial sorting for improved cache locality, macro-based recursion inlining to enhance ILP, and SIMD vectorization using both AVX512 and NEON instructions. Experimental evaluations on both ARM and Intel architectures show substantial performance improvements, achieving up to a $32\times$ speedup compared to a naive baseline implementation.

1. INTRODUCTION

The Barnes-Hut algorithm performs efficient approximation of long-range forces in N-body simulations, which are common in fields like astrophysics (simulating galaxies, dark matter, solar systems), molecular dynamics, and even machine learning (e.g., dimensionality reduction in t-SNE [1] [2]).

Motivation. The naive computation of all pairwise interactions scales quadratically with the number of particles ($O(n^2)$), which becomes infeasible for even modestly large datasets. To overcome this, approximation algorithms such as Barnes-Hut have become predominant in many fields. By organizing particles into a hierarchical spatial data structure (e.g., a quadtree in 2D or an octree in 3D), the algorithm allows distant groups of particles to be treated as single aggregated units, reducing the computational complexity down to on average $O(n \log n)$.

Despite algorithmic efficiency, real-world performance of course also depends on how well this algorithm is implemented. A poorly optimized Barnes-Hut implementation may perform worse than a brute-force method on small to mid-size systems, or may fail to scale due to architectural details like caching issues.

Speeding up Barnes-Hut requires solving some challenging low-level issues: The tree must be built efficiently and in a dense layout, optimized for quick look-ups. Data should be organized in memory to reduce delays from cache misses. SIMD instructions can boost performance, but only if the data is properly aligned. We compare our results to the initial baseline implementation [3] we started from, as well as three other optimized Barnes-Hut implementations [4, 5, 6].

Usual Optimisation Strategy. Most high-performance Barnes–Hut codes chase speed by scaling *horizontally*: They spawn many threads (or GPU kernels) and simply split the particle loop across cores. Parallelism is attractive because the algorithm is inherently parallel at the particle level, so adding hardware threads yields near-linear gains. The downside is that the single-thread kernel often remains nearly identical to the naive textbook version, leaving significant untapped headroom on each core.

Contribution. We present a single-threaded Barnes-Hut C Implementation that is optimized for ILP and cache locality and can utilize both AVX-512 and NEON SIMD instructions. Hence, we focus on squeezing the last cycles out of a single core through microarchitectural optimizations. Depending on the number of particles, our best implementation can achieve a relative speedup of up to $32\times$ to the baseline version. Importantly, our implementation does not rely on any approximations (apart from float arithmetic associativity), therefore delivering the same results as a regular Barnes-Hut implementation, but with a much bigger speedup.

2. BACKGROUND ON THE ALGORITHM

The Barnes-Hut algorithm (Algorithm 1) serves as an efficient approximation of the gravitational forces acting on n particles with predefined masses, initial positions and initial velocities. To speed up these gravitational force calculations, the algorithm sorts the n particles in a three-dimensional space by inserting them into a spatially organized data structure, a so-called octree. Each inner node in the octree then represents all its descendant nodes as one combined mass and its corresponding position (the center of mass). Using this data structure enables the algorithm to not calcu-

Algorithm 1 Barnes-Hut N-body Simulation

```
1: for each time step do
2:     Initialize empty octree
3:     for each particle do
4:         Insert particle into the corresponding sector of the
           octree
5:         Recursively subdivide until there is only one node
           per sector
6:     end for
7:     for each particle do
8:         Traverse the octree to identify nodes satisfying the
           θ criterion
9:         Compute the net force from both internal and ex-
           ternal nodes
10:        Update particle's velocity and position based on
           computed forces
11:    end for
12: end for
```

late all $n \cdot (n-1) \approx n^2$ force contributions between every 2 particles. Instead, the algorithm walks the octree for each particle and for each inner node in the tree checks the so-called Barnes-Hut criterion to decide whether calculating the force contribution of the inner node is a sufficiently good approximation for calculating the force contribution of all its descendant nodes. This reduces the asymptotic runtime of the force calculations between n particles from $O(n^2)$ to $O(n \log n)$ in the average case. We use the most commonly used Barnes-Hut criterion $l/d < \theta$, where l is the side length of the octant represented by the current inner node, d is the distance between the current particle and the inner node's center of mass, and θ is a threshold parameter. Hence, a larger opening angle aborts the tree walk earlier θ (faster but less precise), while a smaller value does the opposite, degenerating to $O(n^2)$ for $\theta = 0$.

To run a particle simulation over a given time, we build the Barnes-Hut tree for each time step and afterward discard it. Trying to reuse it for a later time step is unfeasible, as its construction is by definition highly dependent on the exact particle positions, which change with every iteration.

Hot-Path Analysis. In our optimization effort, we mainly concentrate on the "hot path" of our algorithm - the code consuming most of our CPU time. For Barnes–Hut, this consists of the per-timestep tree traversal for each particle and the force computation for each accepted node-particle pair. All optimisation choices are measured against their effect on this traversal loop. Pre- and post-processing steps in the "cold" sections are acceptable if it shaves cycles off the hot path. Although these operations may incur minor overhead initially, their impact is amortized when scaling to large particle sets, precisely the scenario where Barnes-Hut is typically applied.

Cost Analysis. In this analysis, we focus on the hot-path, as this is where the majority of the computation takes place. A challenge for this algorithm is that the precise amount of flops depends on the distribution of the data and the selected θ, as this directly affects how many nodes within the octree a given particle interacts with. A naive attempt would be to model the cost function as $C_{HP} = m * n * C * log_8(n)$, where n is the number of particles, m the number of simulation steps and C the cost for a singular force calculation. The intuition here being that, on average, each particle interacts with $O(log(n))$ nodes within the octree. However, this results in vastly inaccurate results, in part due to the implicit dependence on θ and the distribution of the particles, which is ignored in this model. We therefore add the coefficient $\alpha(\theta)$: $C_{HP} = m * n * C * \alpha(\theta) * log_8(n)$. This $\alpha(\theta) * log_8(n)$ signifies the actual number of force calculations, compared to the rather low, solely tree-height-based estimate of $log_8(n)$. As our θ remains constant (at 0.5) for all of our subsequent tests, we attempted to estimate its associated α, by sampling the number of times we compute forces per particle for increasing particle sizes (up to 2^{22}). Experimentally, we found that this amount increases in a similar logarithmic fashion, and estimated the value for $\alpha(\theta) \approx 170.55$ using linear regression. This value is inaccurate for smaller n due to the impact of the "cold" sections, but delivers reasonably accurate estimations as n increases, when compared to the actual FLOPs measured.

3. IMPLEMENTED OPTIMIZATIONS

Our goal was to identify and eliminate inefficiencies at both the algorithmic and microarchitectural level. This section describes the optimizations we implemented in the final versions, ranging from mathematical simplifications to low-level memory and computation improvements that together resulted in the final $32\times$ speedup compared to the baseline.

Baseline. The baseline implementation was referenced in the initial task description and was a Python implementation operating in 2D. To use it as a proper comparison point, we translated it into a C implementation and added a third spatial coordinate. Additionally, we built a simple but extensible testbed to support future improvements and experiments. This included functionality for seeded particle generation, simulation and the output of the final positions, which allowed reproducible self-consistency testing. We also added floating point operation (FLOP) counters for performance measurement and analysis.

By the task description, the input and output format was specified to be in polar coordinates. However, we chose to convert all inputs to Cartesian coordinates internally when reading them, as it eventually became apparent that many algorithmic optimizations work best in the latter coordinate system. We still include the conversions in our cycle mea-

surements to ensure a fair comparison.

Trigonometric Optimizations. While first instrumenting our baseline implementations, we quickly outlined some major bottlenecks. First, we found that more than half of the runtime is spent within trigonometric functions. Four of these expensive functions (`sin`, `cos`, `arccos` and `atan2`) are required per particle-node force calculation, which therefore became a reasonable first target for optimization.

Our first attempt replaced the trigonometric functions with a precomputed lookup table of different sizes. Although this partly improved performance, the accumulated numerical error became noticeable after just a few simulation steps. Increasing the granularity of the lookup table again degraded performance.

However, after further investigation, we realized that we could eliminate all calls to trigonometric functions entirely by applying basic algebra and geometric identities. Concretely, rather than computing:

$$\phi = atan2(d_y, d_x)$$

$$\theta = acos\left(\frac{dz}{d}\right)$$

$$\sin\theta = \sin(\theta) \quad \cos\theta = \cos(\theta)$$

$$\sin\phi = \sin(\phi) \quad \cos\phi = \cos(\phi)$$

we instead used:

$$dzd = \frac{dz}{d} \quad dxy = \sqrt{dx^2 + dy^2}$$

$$\cos\theta = dzd \quad \sin\theta = \sqrt{1 - dzd^2}$$

$$\cos\phi = \frac{dx}{dxy} \quad \sin\phi = \frac{dy}{dxy}$$

Flattree. Next, we focused our attention to implement a data structure more tailored towards tree traversal than the initial, pointer based tree. Concretely, we created a flattened, struct-of-array-based tree representation we refer to as *FlatTree*, to improve memory locality and eliminate the overhead of pointer-based traversal ("pointer chasing"). This FlatTree struct includes arrays holding the positions and masses for all tree nodes, as well as a `child_index` used as an index into the tree to where the set of eight children for this node are located. We also use this tree to precompute any repeatedly used values - eg. the squared sector side length, or wether this node is an inner, outer or empty node - and at the same time remove all values not explicitly required for tree traversal to avoid unnecessary filling up the cache.

We initially also decoupled tree traversal from force computation here, but ultimately removed this approach due to its added overhead; this variant is discussed in Section 4.

Iterative Tree Traversal. Initially, we used a simple recursive approach to traverse the Barnes-Hut tree. However, recursive function calls are very costly performance-wise if not optimized away by for example tail recursion, which is not possible here. This is in part due to method arguments being passed through the stack, as well as ILP scheduling not being performed across call instructions. To address this, we adopted an explicit iterative traversal strategy using a hand-unrolled stack. Specifically, we maintain a stack of integer node indices which correspond to entries in our FlatTree array. This approach eliminates the overhead of recursive calls.

To replicate the depth-first search (DFS) traversal - visiting child 0 first and proceeding in increasing order - we push the child nodes onto the stack in reverse order (i.e., from child 7 down to child 0). This ensures that when nodes are popped in LIFO order, they are processed from child 0 upward. We also evaluated breadth-first search (BFS), but found DFS to be slightly more performant, matching the DFS order in which our FlatTree is encoded.

Sorting the particles. To further improve memory locality, we introduced a sorting step that reorders particles and therefore traversal order based on spatial proximity. The intuition here is that nearby particles are likely to hit similar areas in the FlatTree during traversal. By processing them in an order where every successive pair of particles is sufficiently "close" to each other, we significantly reduce the amount of (higher-level) cache misses within the FlatTree, especially for larger particle sizes.

In order to sort the particles based on proximity, one can use the Barnes-Hut tree itself. By construction, the parent-child relationships encode proximity information for all particles: For every node, its children are completely encompassed into its bounding box and therefore overall closer to each other than to any other node. Due to these relations, traversing the tree using DFS visits the outer nodes in an order similar to a Peano-Hilbert curve. Sorting the original particle array based on this curve (which determines the processing order) serves as a sufficiently good proximity heuristic to see major runtime improvements, especially in larger-scale simulations. In order to amortize the performance impact of having to fully traverse both the particle array and Barnes-Hut tree, we also experimented with different intervals for sorting (eg. only sort every sixteen timesteps). However, manual testing quickly showed that sorting every timestep, directly after constructing the tree, is the fastest variant.

We also briefly explored a traversal-splicing technique for further cache-locality gains (see Section 4), but found its overhead outweighed any benefit.

NEON/AVX-512. After completing the main batch of scalar optimizations, we opted to translate the established scalar design paradigms into vectorized implementations. As we had both Intel and ARM processors available to us, we were interested in how these platforms would compare

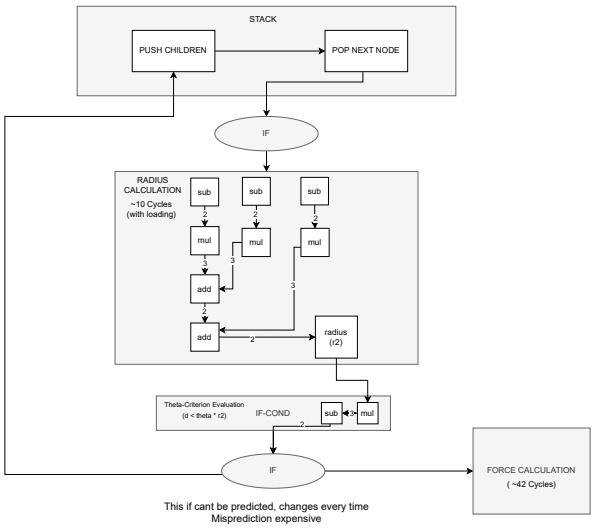

Fig. 1: Stack-based traversing of the tree is inherently limited by a cyclic dependency

in terms of SIMD performance. We therefore opted to implement vectorized versions using both NEON (for ARM) and AVX-512 (for x86). Especially AVX-512 felt like a natural choice for a 3D Barnes-Hut implementation, due to the structure of the octree: each internal node has exactly 8 children, which aligns perfectly with the 512-bit width of AVX-512 registers (8 × 64-bit doubles). This allowed us to process all child nodes of an inner node in a single SIMD instruction pipeline. On ARM, NEON registers support only two 64-bit doubles per vector, so we mirrored the AVX-512 strategy by processing pairs of children sequentially in 2-wide lanes. One inherent difficulty here is handling the Barnes-Hut acceptance criterion in SIMD: different children within a vector might satisfy the θ test differently, therefore requiring divergent control flow - either computing the force or pushing to the stack. We evaluated different approaches like selective computation via gathering, but found that unconditionally computing all eight force interactions and then using a vector mask to blend results yielded the best speedup.

Instruction-Level Parallelism (ILP). Following feedback during our second meeting with the head TA, we realized that stack-based tree traversal always causes a cyclic dependency. Specifically, the traversal involves popping a node from the stack, checking whether it satisfies the Barnes-Hut θ criterion, or pushing its 8 children onto the stack. The if-condition on the θ check is especially prone to performance degradation due to this cyclic dependency, as it either stalls the instruction pipeline or provokes branch prediction on an inherently unpredictable branch. To address this, we experimented with different variants of increasing ILP. Our key insight was that for the else branch, instead of pushing the children to the stack, one could simply inline the "recursive call" for every child. Having this function body placed 8 times in sequence (one for every child) allows the processor to better schedule independent instructions, as while waiting for the result of one if condition, the data can be loaded and computed for the next. This can also be performed recursively, inlining 8 times for children, 64 times for grandchildren, etc. We implemented this by using layered C macros, where eg. macro `L1` calls macro `L2` 8 times, expanding the logic for each of the 8 child nodes. When reaching the deepest layer, the algorithm falls back to pushing onto the stack, retaining but effectively loosening the problematic cyclic dependency. We experimented with varying levels of inlining, tuning the depth to balance the opportunity for ILP with the incurred instruction cache pressure. Fine-tuned for both our ARM machines and Intel Skylake AVX-512 evaluation server, the final version had 2 levels of inlining - resulting in approximately 45,000 lines of expanded C code - and provided significant performance improvements over our prior scalar baseline.

SIMD ILP. Encouraged by the performance gains from our scalar ILP-optimized implementation, we extended the same strategy to our SIMD versions. The goal was to combine vectorization with the ILP unrolling to maximize both data-based and instruction-level parallelism. To achieve this, we again used macros to unroll the tree traversal logic, fine-tuning the level of inlining for each target architecture. After further evaluation, we found the optimal unrolling level of two for both ARM (NEON) and Intel Skylake (AVX-512), as with the scalar variant.

By merging vectorization and ILP inlining, we achieved our largest overall speedup, up to $32\times$ faster compared to our original baseline implementation.

4. UNPRESENTED OPTIMIZATIONS

Splitting. As one of the first tasks, we applied a presumed "optimization" we called splitting: During traversal, each particle filled a buffer that stored the x, y, z coordinates and mass of every node passing the θ test; immediately afterward the same particle consumed this buffer again to compute the forces. The idea was twofold: First, the actual force then contained no branches whatsoever, which would simplify optimizing for ILP and later for a SIMD kernel; second, we assumed the simulation was mainly compute-bound and could therefore absorb the extra memory traffic. In reality, the additional writes and reads increased cache pressure quite a bit, shaving performance off *every* variant in which it appeared. Inlining the force evaluation back into the traversal also let us reuse some intermediate values such as distances and their inverses. We therefore removed splitting from all versions that initially included it, which lead to

the implementations we presented above.

FlatTree condensing. Another design decision revisited later concerned the construction of the FlatTree. To simplify handling of children and to keep them aligned to 64-byte boundaries, we force every inner node to have exactly eight child entries, inserting dummy leaves for missing octants. However, profiling a medium-sized tree showed that more than half of all visited nodes were such empty placeholders, inflating memory traffic and polluting the cache. We therefore tested a condensed FlatTree that stores only the children actually present, ignoring alignment for the time being. This sparse layout gave a noticeable boost for the plain FlatTree variant. Yet, when integrated with the ILP and SIMD variants the benefit vanished: The variable child count re-introduced branches and loops that slowed down the otherwise straight-line code paths. After extensive evaluation of variants trying to bypass this issue (again resorting to macros and switch-case statements) we retained the original sparse eight-child layout, accepting the wasted loads in exchange for branch-free(er) traversal, as it gave a better overall performance.

Traversal Splicing. Early during our optimization attempts we noticed the inherent difficulties with efficient memory access when traversing a tree-like structure. For this purpose, we attempted to implement a method called "Traversal Splicing" introduced by Jo et al. [7], aiming to improve memory locality for tree traversal. The basic idea was to split a tree traversal into a "top" and "bottom" part. During the "top" part, the tree is traversed by all particles up to a specific depth, before stopping. We then, for each particle, collect the node where an individual particle is "waiting". Under the assumption that particles which have followed a similiar traversal up to depth D will behave similarly afterward, we then sort the particles by the nodes they have ended their "top" traversal at, and then perform the "bottom" traversal in that order. Ideally, this increases locality because we traverse the same section of our data structure multiple times, decreasing cache misses. What is important, however, is that this approach requires frequent sorting after every "top" traversal, meaning that (even when exploiting that we can use sub-$n \log(n)$ algorithms like CountSort here) this adds a lot of overhead. In the end, we found that this approach (for the evaluated particle sizes) performed similarly to our previous implementation, which relied only on sorting as discussed above. Performance actually decreased with higher particle numbers, as the sorting took progressively more time. We therefore stuck with our initial approach and decided not to pursue traversal splicing further.

MIPP. As our team was working in parallel on both ARM and x86 machines, we were looking for a way to make our code more portable - not essentially with performance gains in mind, but to enable quick, iterative prototyping of SIMD optimizations (like ILP). The MIPP library caught our attention for this purpose, as it provides an API that automatically translates code written using MIPP instructions into a particular machine's SIMD instructions. It relies heavily on operator overloading, producing easy-to-understand code that looks very similar to a scalar implementation. MIPP required a bit of work to deliver consistent results and function without errors (concretely, using a approximate `sqrt` instruction on NEON without explicitly acknowledging it). However, after resolving these issues, and to our surprise, the MIPP-AVX-512 version ended up performing quite a bit faster than our native AVX-512 implementation. This in turn led us to investigate our original code, finding multiple areas for improvement, which led to a considerable speedup of our final results.

Although the presented optimizations do not use any MIPP code - in summary, all of them perform slightly worse than the respective native implementations - the library proved to be very useful for prototyping and validating them.

5. EXPERIMENTAL RESULTS

In this section, we present the performance and runtime improvements achieved by our optimizations. First, we discuss the experimental setup and the results of each optimization step we performed. Finally, we reason whether our optimizations are close to the optimum for implementing Barnes-Hut.

Experimental Setup. Our experiments were conducted on an Apple M1 Max system (for NEON) and a remote server using the Intel Skylake architecture (for AVX-512). However, since running the measurements for large inputs takes considerable time and we cannot completely remove all interference from other applications on our Apple M1 Max machines, we focus on the Intel Skylake results. We conducted the experiments on an Intel Xeon Gold 6132, providing $896\,\mathrm{KiB}$ L1 and $28\,\mathrm{MiB}$ L2 cache per core and $38.5\,\mathrm{MiB}$ shared L3 cache. We compiled our implementations using `gcc` version 12.2.0 and used `-O3 -fno-tree vectorize -ffast-math -march=native` as our compile flags.

The compilers and compiler flags mentioned above represent our final configuration. During optimization, we also tried different compilers and altering the compiler flags. However, we observed similar results for `clang`, `gcc`, and `icc`. Furthermore, optimization flags beyond the above-mentioned did not yield any noticeable performance or runtime improvements. We disabled auto-vectorization to ensure clean separation of scalar and SIMD code in our evaluation.

To evaluate the performance and runtime of our optimized implementations, we generated random particles in 3D space. We scale the number of particles exponentially from 2^4 to 2^{22} (only up to 2^{17} on Apple M1 Max). Finally,

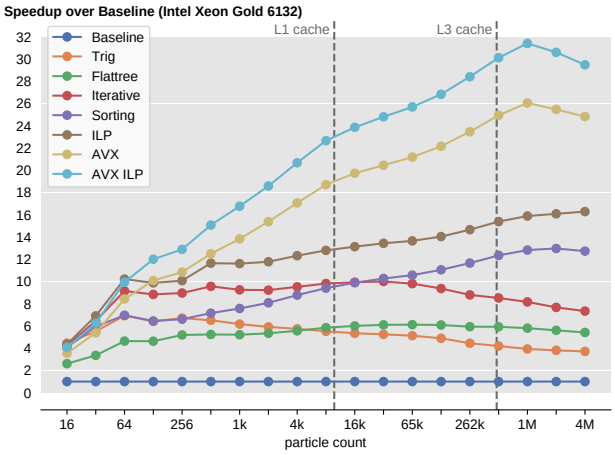

Fig. 2: Speedups achieved by all our optimized implementations compared to our baseline.

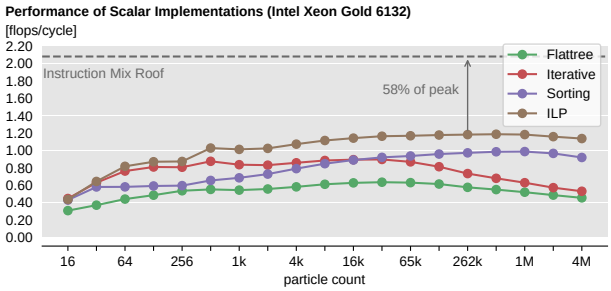

Fig. 3: Performance of our scalar implementations in relation to the instruction mix roof.

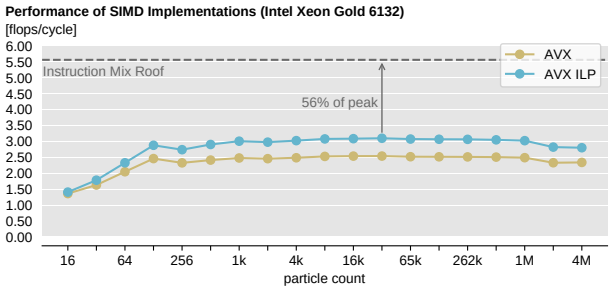

Fig. 4: Performance of our AVX implementations in relation to the instruction mix roof.

we run the Barnes-Hut simulation for 50 time steps to capture the algorithm's performance characteristics regarding the changing particle distribution.

To measure the FLOPs of our algorithm, we manually instrumented the code with FLOP counters. We use the `TSC` (Time Stamp Counter) on Intel and `PMC` (Performance Monitoring Counter) on Arm to measure the cycles. Before measuring the cycles, we warm up the cache by performing one step of the Barnes-Hut algorithm before starting the 50 steps we measure. Furthermore, we ensure sufficient cycles (by performing multiple runs) even for small inputs to get reliable results. We run two different versions of the binary for the FLOP and cycle measurements to avoid interference.

Results. We present the speedups of the optimizations explained in Section 3 in Figure 2. Besides the speedups, we also include the particle count thresholds where the entire FlatTree still fits into the L1/L3 cache.

First, we observe that replacing the expensive trigonometric functions already reduces our runtime by a factor of 4-6×. Second, by removing the pointer chasing associated with the primitive octree and pre-computing reusable values, we achieve a small speedup for larger inputs in the FlatTree version. For small inputs, the overhead of constructing the FlatTree is larger than the memory benefits it provides. Third, the iterative version introduces a significant runtime reduction for all input sizes, peaking at 10× lower runtime over the baseline runtime. We attribute this speedup mostly to the removed recursive function call overhead and the additional compiler optimizations enabled by the iterative version (e.g., the tree traversal can be inlined). Fourth, the increased temporal locality introduced by particle sorting reduces the overall runtime for large inputs (> 32k particles). Furthermore, the sorting reverses the trend of previous implementations in that the speedup decreases for large inputs (> 32k particles). This trend reversal can be explained by the fact that we mostly access L1-resident data during the tree traversal by using sorting. Even up to the input size of four million particles, we access only about three to four thousand FlatTree nodes during the traversal. This number of nodes easily fits into the L1 cache, and the spatial sorting ensures that we mostly access previously accessed FlatTree nodes. This effect can be observed in the plot since the speedup of the iterative traversal drops compared to the sorting variant once the entire FlatTree no longer fits into the L1 cache. Furthermore, we notice in the roofline plot (cf. Figure 5) that the sorting stabilizes the operational intensity of the algorithm. Finally, the stack unrolling introduced in the ILP version again reduces the algorithm's runtime, reaching our maximum improvement for scalar code at 16× over the baseline implementation.

For the SIMD code (AVX and AVX ILP), we observe a significant speedup over the scalar code for larger inputs (> 1k particles), reaching our maximum speedup over the baseline with 32× for AVX ILP. As in the scalar code, we observe a significant runtime decrease from unrolling the tree traversal stack. We attribute the speedup drop after the peak at 1 million particles to the size of the L3 cache. We argue that the drop is delayed by the improved access pattern introduced by the spatial sorting.

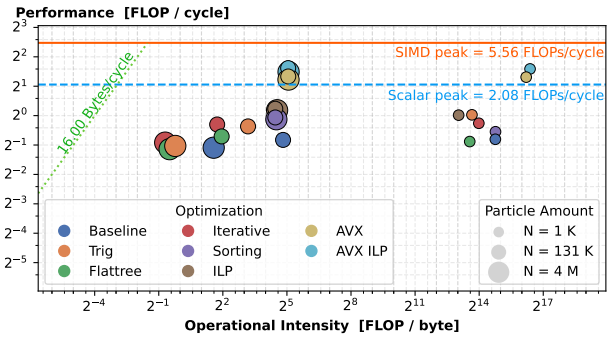

Fig. 5: Roofline plot for the various optimizations at selected sizes

Regarding the performance of our implementations, we split the evaluation in scalar (cf. Figure 3) and SIMD variants (cf. Figure 4) since the SIMD variants increase the FLOP count compared to the scalar variants. In the scalar case, we observe that the Sorting and ILP optimizations boost performance especially for large inputs. For smaller inputs, we also see a significant performance impact by replacing the recursive traversal. Overall, our ILP variant reaches a peak performance of about $1.2\,\mathrm{flops/cycle}$. Compared to the theoretical peak of a Skylake core of $8\,\mathrm{flops/cycle}$, this performance is dissatisfactory. However, when considering the instruction mix, we can derive a lower roof of approximately $5.56\,\mathrm{flops/cycle}$. We obtain this roof by analyzing that about $8\,\%$ of our FLOPs are square root operations that have a throughput of $1/6$ on Skylake. Hence, we need at least $48\,\mathrm{cycles}$ for $100\,\mathrm{flops}$, leading to a theoretical peak of $2.08\,\mathrm{flops/cycle}$. With respect to this improved bound, we reach $58\,\%$ of peak performance.

In the SIMD case, we observe a performance increase by a factor of about 2.5. Again, the ILP optimization significantly boosts the performance of the algorithm. Similar to the scalar case, the peak performance of $3.1\,\mathrm{flops/cycle}$ is nowhere close to the theoretical peak of $32\,\mathrm{flops/cycle}$ achievable using AVX-512 on Skylake. However, we can again derive a lower bound, yielding a new theoretical peak of $5.56\,\mathrm{flops/cycle}$ based on the square root operations. Using this improved bound as the reference, we achieve $56\,\%$ of theoretical peak performance.

Overall, we observe that the instruction mix significantly limits us and that we are not bound by memory accesses (cf. Figure 5). To overcome this issue, we experimented with approximate inverse square root instructions where possible (e.g., `_mm512_rsqrt14_pd`). However, in many cases, we cannot use such instructions since small inaccuracies might already lead to a division by zero.

For the NEON implementations, we did not observe any significant benefit of using SIMD over our scalar ILP implementation. For input sizes of about 128k particles, we measure a speedup of about $17\times$ over the baseline for both ILP and NEON ILP. However, we observe about $1.5\times$ higher performance for NEON ILP compared to the scalar ILP version, indicating that we perform some needless work due to the vectorization (i.e., computing forces on empty nodes in the FlatTree).

Comparison. Finally, we compared our fastest implementation AVX ILP to three other optimized implementations [4, 5, 6]. We first extracted the core Barnes-Hut algorithm, since some implementations included multi-processing support or particle simulations. Then, we measured the algorithms' runtime on a single core for 1M particles. We compiled with the same flags we used for our evaluation where possible. Our measurements show that our optimized AVX ILP implementation is faster than all three variants we tested against. We achieve $24\times$ faster runtime over the OpenMPI implementation [5], which is expected since this variant is mainly optimized for multi-core processing. However, we furthermore achieve $3.3\times$ over [4] and $2.7\times$ over [6], which both claim to be optimized for running on a single core.

6. CONCLUSIONS

We have demonstrated that a microarchitectural-aware approach to a classic algorithm can yield dramatic performance gains, even in a single-threaded setting. By eliminating expensive trigonometric calls, replacing a pointer-based octree with a cache-friendly FlatTree, transforming recursion into an explicit DFS stack, introducing spatial sorting, perform ILP-focused inlinings and implement both NEON and AVX-512 vectorization, our final AVX-512-ILP implementation reaches up to a $32\times$ speedup over the textbook baseline, without altering the mathematical fidelity of the algorithm. Moreover, it achieves over 55% of both scalar and vector instruction-mix rooflines on Skylake, underscoring how much performance is accessible without resorting to thread-level parallelism.

These findings underscore two key takeaways: first, that significant headroom exists in mature algorithms when viewed through the lens of modern CPU features; and second, that a highly optimized single-core kernel provides a firmer foundation for scalable parallel variants than a textbook scalar reference ever could. By delivering this high-performance, drop-in reference implementation, we enable astrophysics, molecular dynamics and dimensionality-reduction algorithms (e.g. t-SNE) to achieve greater accuracy and throughput on modest hardware, and provide a solid foundation for future parallel extensions.

7. CONTRIBUTIONS OF TEAM MEMBERS

This team project was split up as outlined below:

Berndt. Implemented the AVX-512 for both non-ILP variants. Implemented Traversal Splicing as described by Jo et al. [7]. Worked on the Cost analysis and integrated PAPI to measure memory traffic and construct the Roofline plot.

Daniel. Worked with Berndt on lookup tables implementation and later on the translation from NEON to AVX-512. Worked on the prototype for AVX2 variant.

Emanuel. Worked with Niels on trigonometric replacements; implemented iterative DFS tree traversal, particle sorting; prototyped and implemented different ILP variants, evaluated force calculation splitting, condensing the FlatTree, primitive tree splicing concept; included and fixed MIPP library.

Niels. Worked with Emanuel on the design and implementation of the FlatTree data structure, implemented the Arm NEON version, worked with Berndt on porting the NEON code to AVX-512, ported the ILP version to work with Arm NEON and AVX-512, tested alternative approximate instructions for faster square root or division operations for both NEON and AVX, experimented with Emanuel on alternatives to the stack unrolling to improve ILP

8. REFERENCES

[1] Laurens van der Maaten, "Accelerating t-sne using tree-based algorithms," *Journal of Machine Learning Research*, vol. 15, no. 93, pp. 3221–3245, 2014.

[2] "GitHub - lvdmaaten/bhtsne: Barnes-Hut t-SNE — github.com," `https://github.com/lvdmaaten/bhtsne`, [Accessed 18-06-2025].

[3] "GitHub - ntta/barnes-hut-algorithm: The algorithm to compute the force on a particle by using quadtree — github.com," `https://github.com/ntta/barnes-hut-algorithm`, [Accessed 18-06-2025].

[4] "GitHub - Canleskis/particular: N-body simulation library written in Rust featuring BarnesHut and GPU accelerated algorithms. — github.com," `https://github.com/Canleskis/particular`, [Accessed 18-06-2025].

[5] "GitHub - barkm/n-body: Barnes Hut implementation using OpenMPI," `https://github.com/barkm/n-body`, [Accessed 20-06-2025].

[6] "GitHub - MarkVivas/N-body: Optimized sequential Barnes Hut implementation including OpenGL visualization," `https://github.com/MarcVivas/N-body`, [Accessed 20-06-2025].

[7] Youngjoon Jo and Milind Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," `https://engineering.purdue.edu/~milind/docs/oopsla12.pdf`, [Accessed 18-06-2025].