N-Body simulation using the Barnes-Hut method

Team 21:

- Berndt Uhlig
- Emanuel Mairoll
- Niels Pressel
- Daniel Baciu

Barnes Hut Algorithm

For all time steps run:

- Build octree (insert particles one by one)
  - Insert in the corresponding sector
  - Recursively subdivide until only one node per sector
- For each particle do:
  - Traverse the BHTree until satisfying the theta criterion
  - Calculate the force to these internal and external nodes of the tree
  - Update particle positions

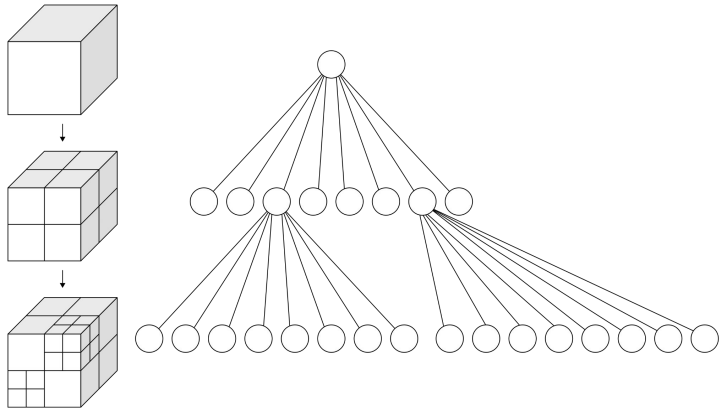


Image source: https://en.wikipedia.org/wiki/Octree#/media/File:Octree2.svg

Determine the Hot Path

For all time steps run:

- Build octree (insert particles one by one)
  - Insert in the corresponding sector
  - Recursively subdivide until only one node per sector
- For each particle do:
  - **Traverse the BHTree until satisfying the theta criterion**
  - **Calculate the force to these internal and external nodes of the tree**
  - Update particle positions

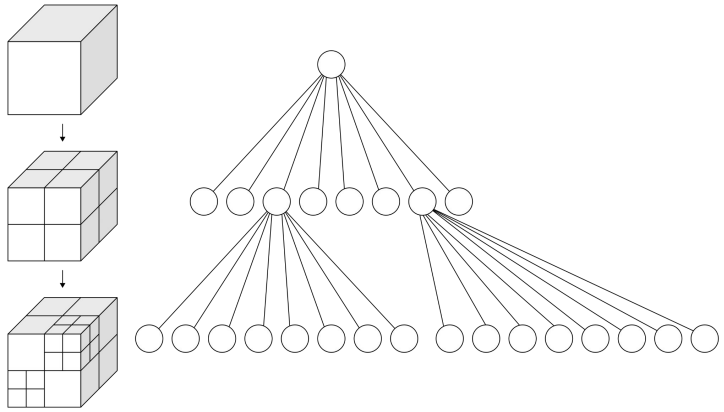


Image source: https://en.wikipedia.org/wiki/Octree#/media/File:Octree2.svg

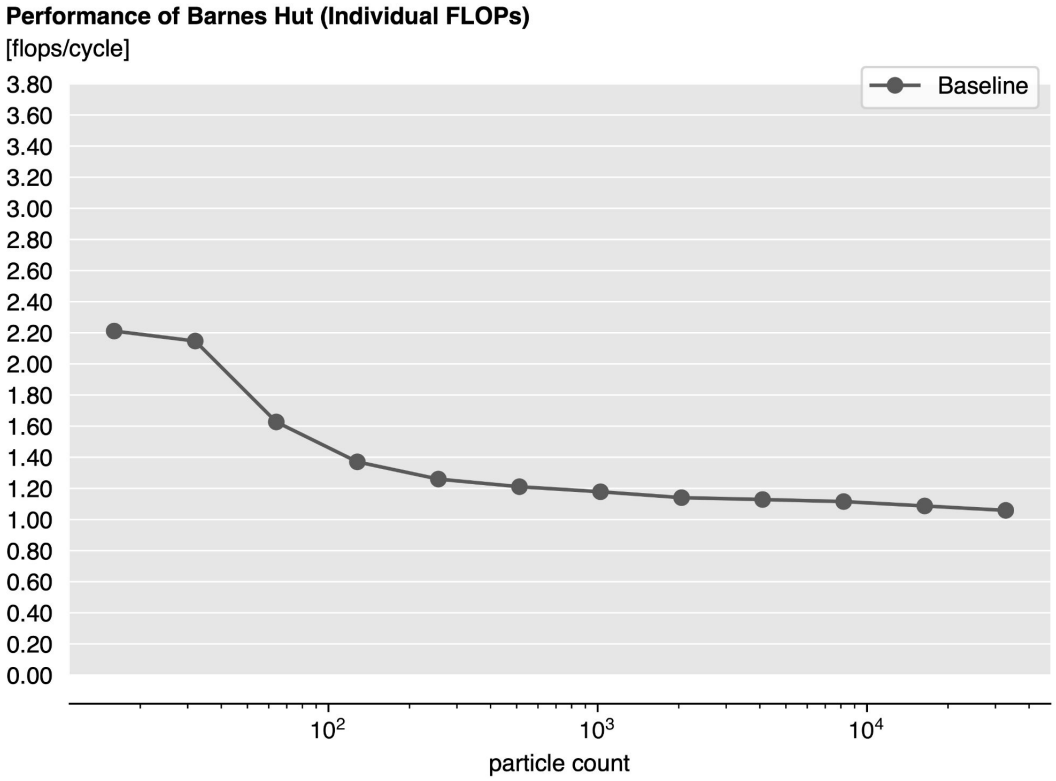
Baseline

```
theta: 0.5
steps: 365
gravity: 6.67430e-11
coord_system: polar

particles
1.989e30, 0, 0, 0, 0, 0, 0          # Sun
3.285e23, 0.387, 0, 0, 0, 4.36, 0   # Mercury
4.867e24, 0.723, 0, 0, 0, 3.02, 0   # Venus
5.972e24, 1.000, 0, 0, 0, 2.78, 0   # Earth
```

Followed the following process:

- Ported, extended, and cleaned up the 2D Python version
- Generate particles or read them from input
  - Convert to Cartesian immediately, is better to optimize
- Instrumented the code with flop counters and added cycle measurements
- However:
  - Linked Data Structures
  - Recomputing reusable values
  - Expensive Operations (ifs, trigonometric, etc…)

Baseline - Plot



Optimizations

Data Structures and Force Calculation

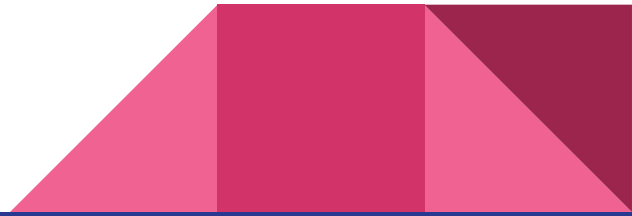
- Linear Tree Representation - “FlatTree”
  - Already with SIMD in mind
  - Struct of Arrays
- Precompute cacheable values
- Split up traversal and force calculation
  - Intermediate dense buffer for straight forward force calculation
  - Also with SIMD in mind
- Keep track of Particle index (p_idx)
  - Stores info about node type (INNER, EMPTY, etc.)
- Remove *all* trigonometric functions

```
typedef struct {
    double *restrict posX;
    double *restrict posY;
    double *restrict posZ;
    double *restrict mass;
    double *restrict maxD;
    int32_t *restrict p_idx;
    // index of first child
    // (-1 -> no child)
    int32_t *restrict first_child;
} FlatTree;
```

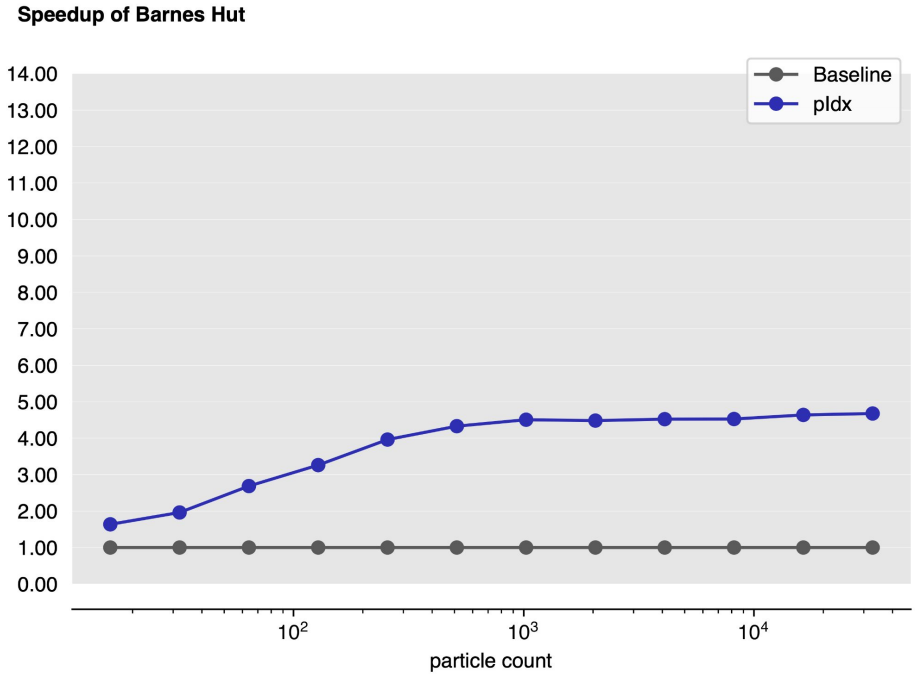
Trigonometric Functions

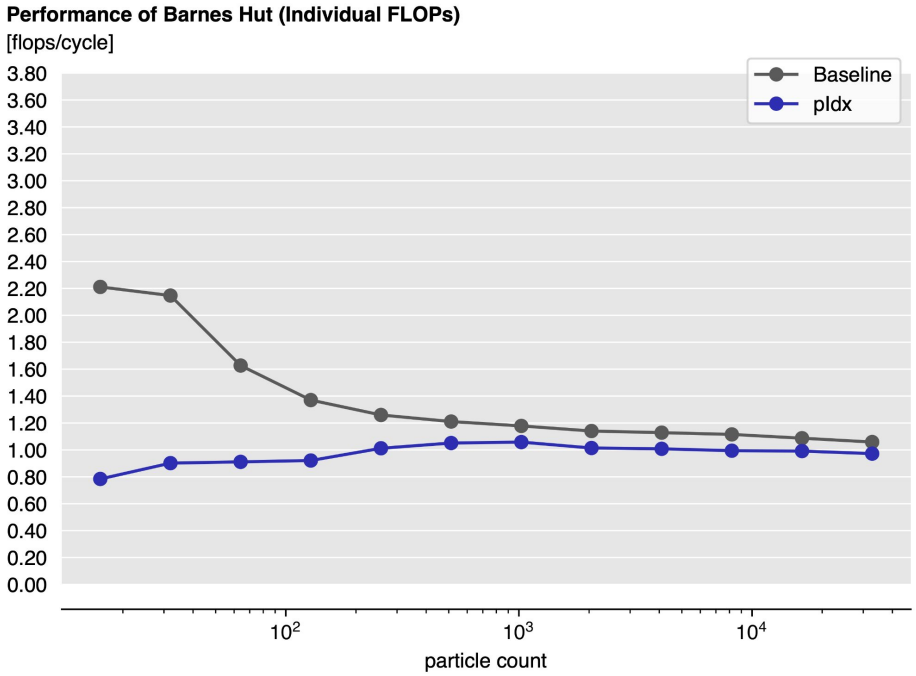
```
double phi = atan2(dy, dx);
double theta = acos(dz / d);
double sinT = sin(theta);
double cosT = cos(theta);
double cosP = cos(phi);
double sinP = sin(phi);
```

```
double dzd = dz / d;
double dxdy = sqrt(dx*dx + dy*dy);
double sinT = sqrt(1.0 - dzd*dzd);
double cosT = dzd;
double cosP = dx / dxdy;
double sinP = dy / dxdy;
```



Data Structures and Force Calculation - Plot



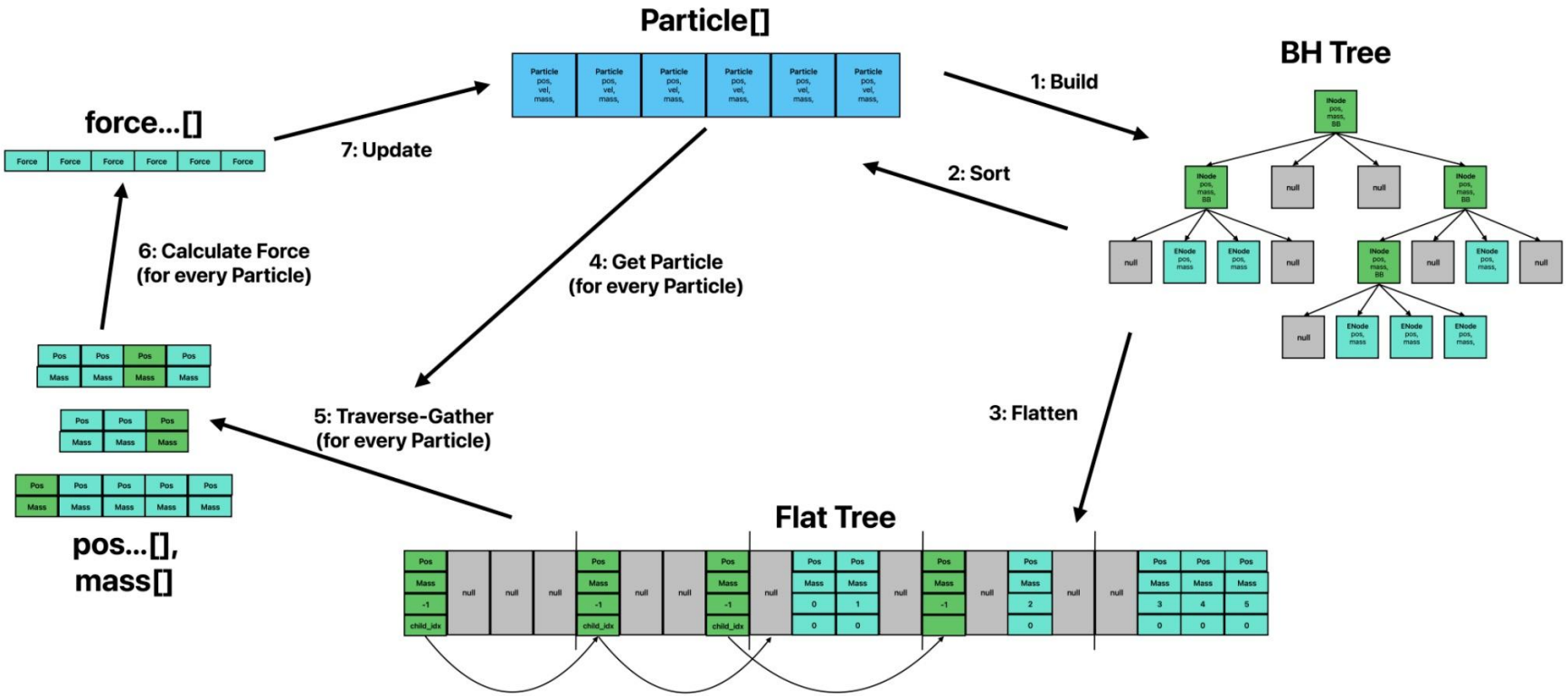


Improving Cache Locality

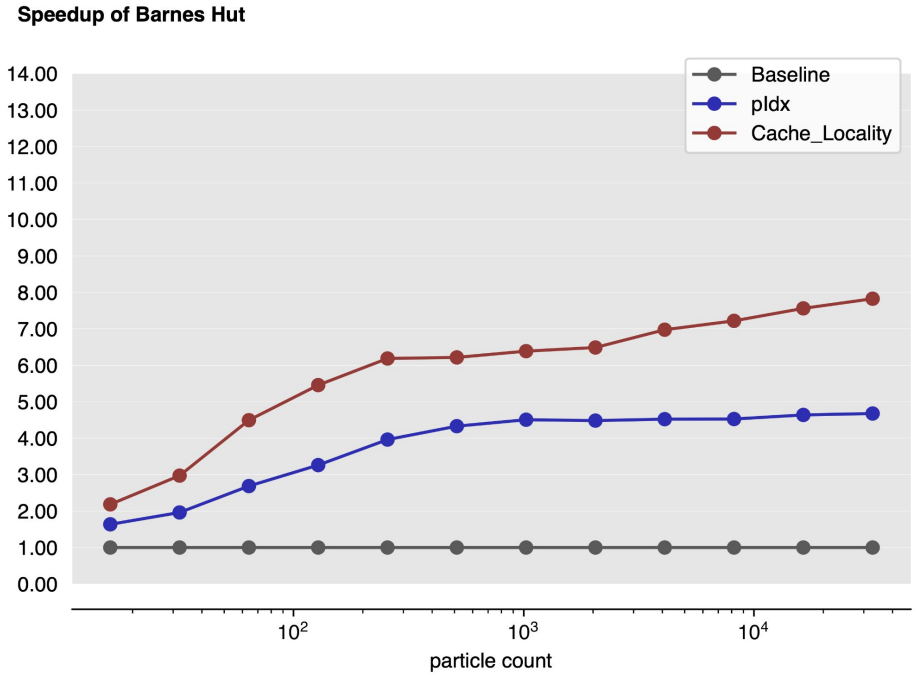
- Switch to iterative traversal to avoid recursion overhead
- Trying BFS/DFS approaches to improve the cache locality
  - Iterative DFS with a stack is the fastest
- Introduce Particle sorting by proximity (using BH tree)
  - For memory locality during traversal
  - Trying sorting every n steps, decided to sort every step

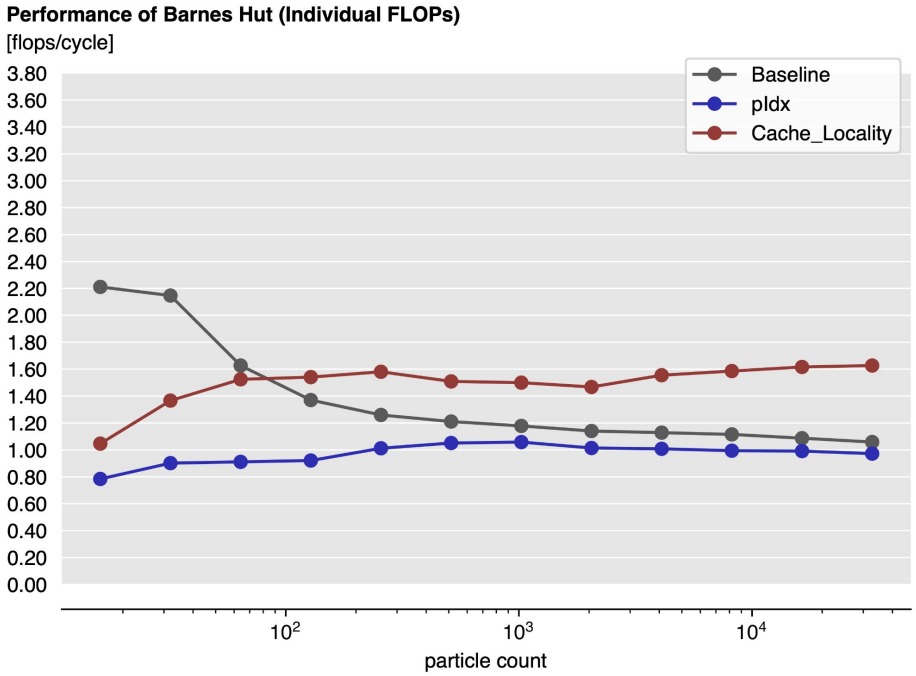
```
// after building BHTree
Particle* tmp = malloc(n * sizeof(Particle));
memcpy(tmp, particles, n * sizeof(Particle));

sort_array_with_tree(root, tmp, particles);
free(tmp);
// then build FlatTree
```

Data Structure Interactions



Improving Cache Locality - Plot



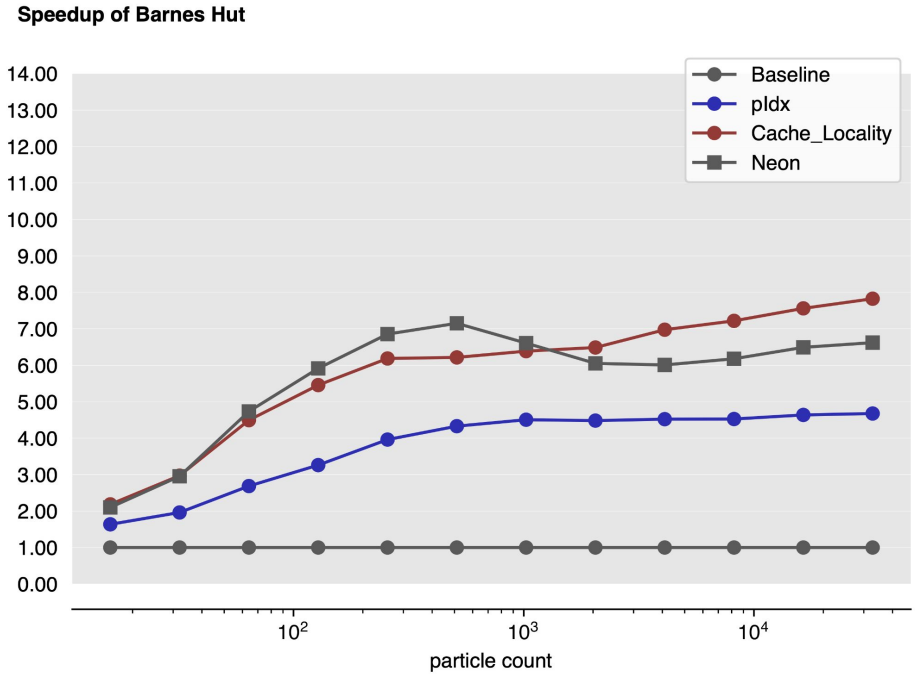


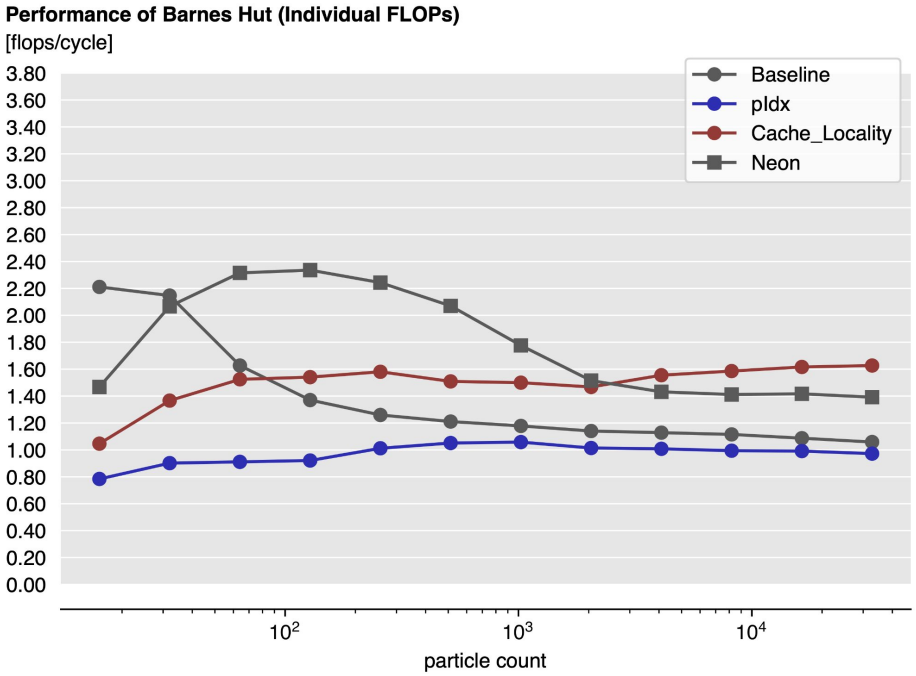
Introducing SIMD (ARM NEON & AVX512)

- Implemented both ARM NEON and AVX512
  - To experiment with different hardware
- Still using recursive tree traversal
  - This optimization step was performed in parallel with the iterative variant
- Duplicate code for Neon caused by smaller register size
  - Compute all 8 children at once
- Avoid ifs where possible
  - Unavoidable for traversal
- Use dense copyout buffer for SIMD force calc

```
float64x2_t fst_part_0 = vfmaq_f64(dx2_0, dy_0, dy_0);
float64x2_t dist_0 = vfmaq_f64(fst_part_0, dz_0, dz_0);
float64x2_t d2_theta_0 =
             vmulq_f64(other_d2_0, theat2_recip_vec);
uint64x2_t theta_crt_0 = vcltq_f64(d2_theta_0, dist_0);
```

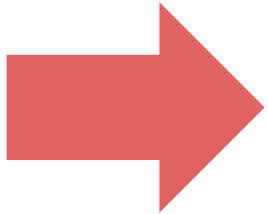
Introducing SIMD (ARM NEON & AVX512) - Plot





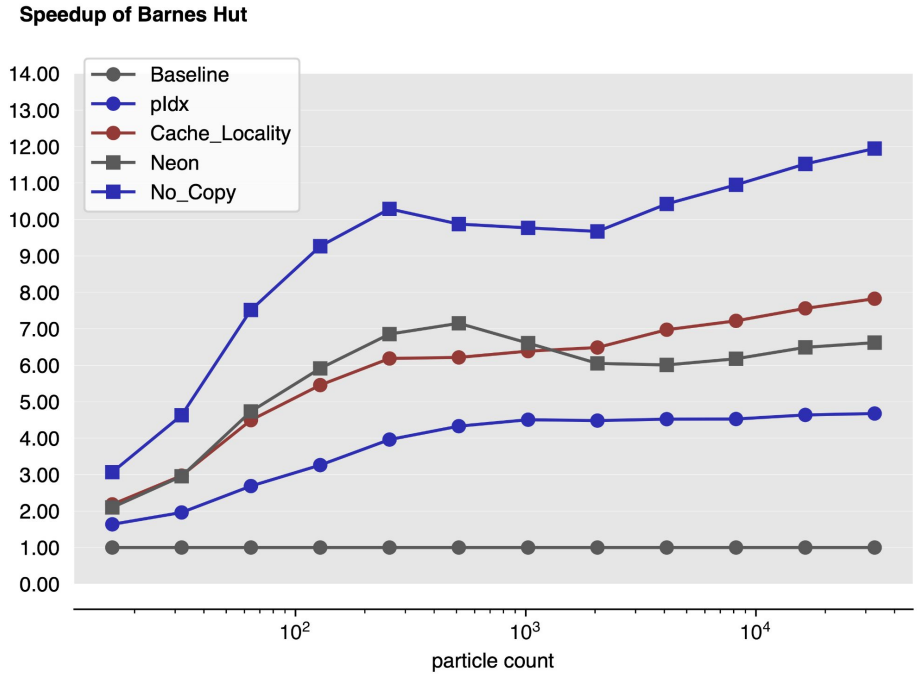
Inlining the Force Calculation (again)

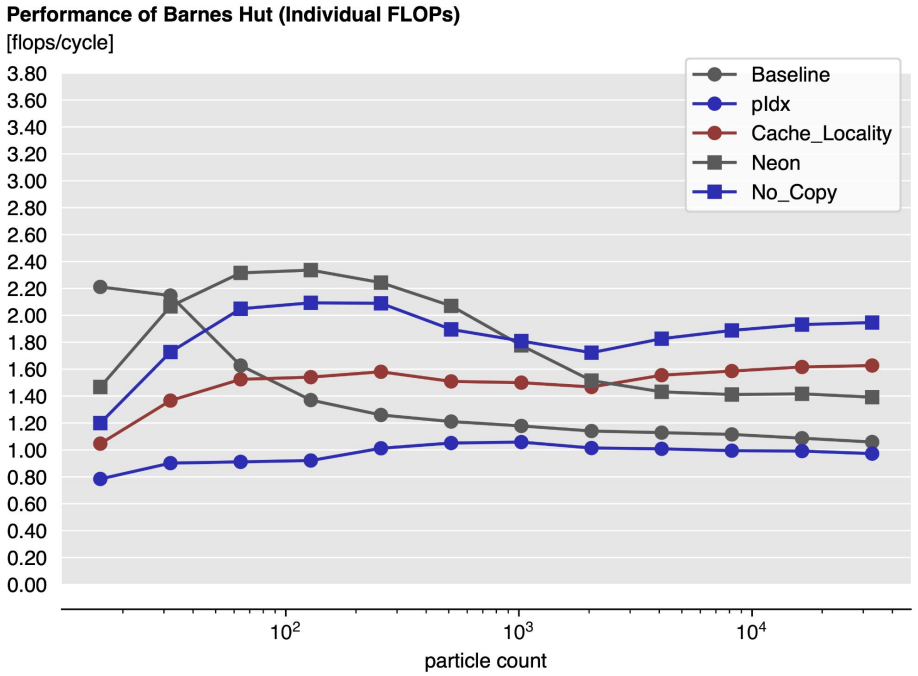
```
for (int32_t i = 0; i < n; i++) {
     int count = 0;

     traverse_gather(....);

     for (int j = 0; j < count; j++) {
          double out_x = 0.0;
          double out_y = 0.0;
          double out_z = 0.0;
          calculcate_force(...);

          fx[i] += out_x;
          fy[i] += out_y;
          fz[i] += out_z;
     }
}
```



```
for (int32_t i = 0; i < n; i++) {
     double out_fx = 0.0;
     double out_fy = 0.0;
     double out_fz = 0.0;

     traverse_calc_force(...);

     fx[i] += out_fx;
     fy[i] += out_fy;
     fz[i] += out_fz;
}
```

Inlining the Force Calculation - Plot

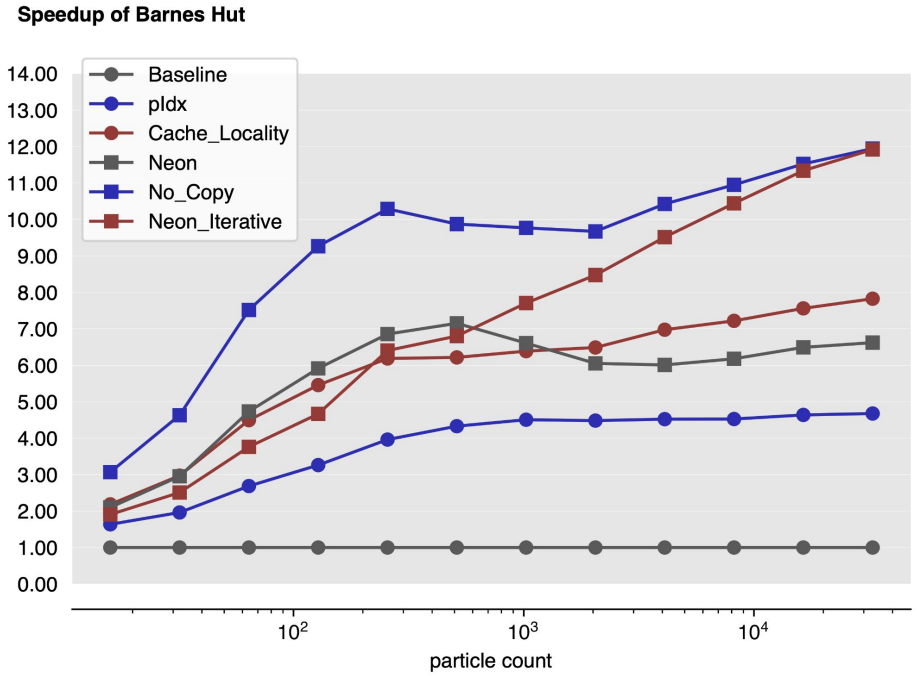


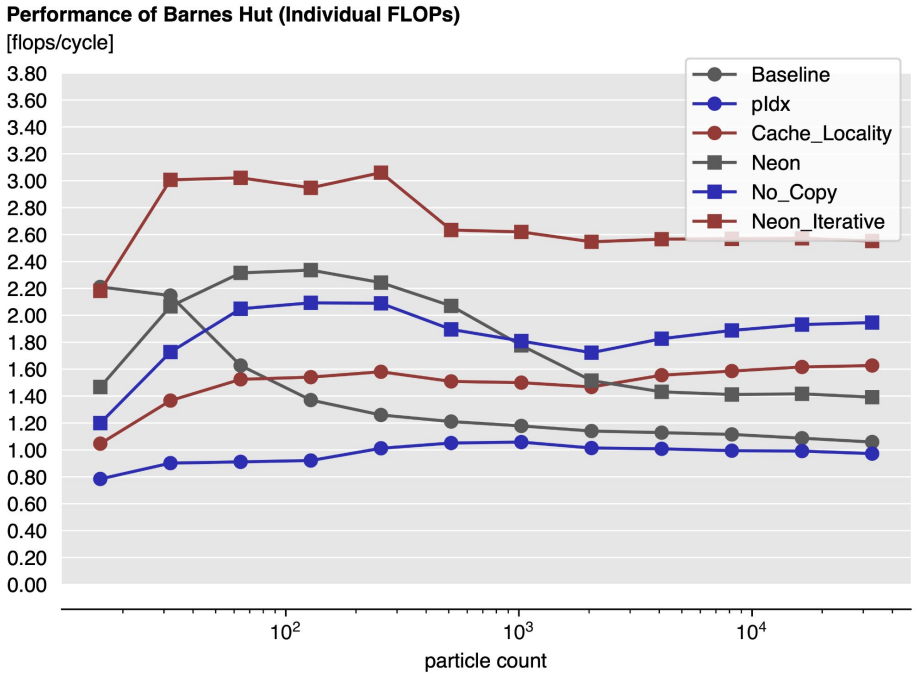


Port to AVX/NEON

- Change to iterative traversal
- Switch from copyout buffer to inlined force calculation
  - Allows us to reuse calculations from the tree traversal
- Avoiding ifs requires calculation of the force for all children
  - Regardless of whether the calculation is needed

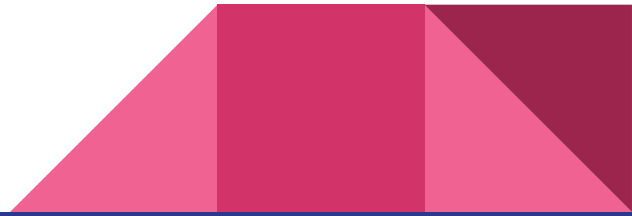
Port to AVX/NEON - Plot



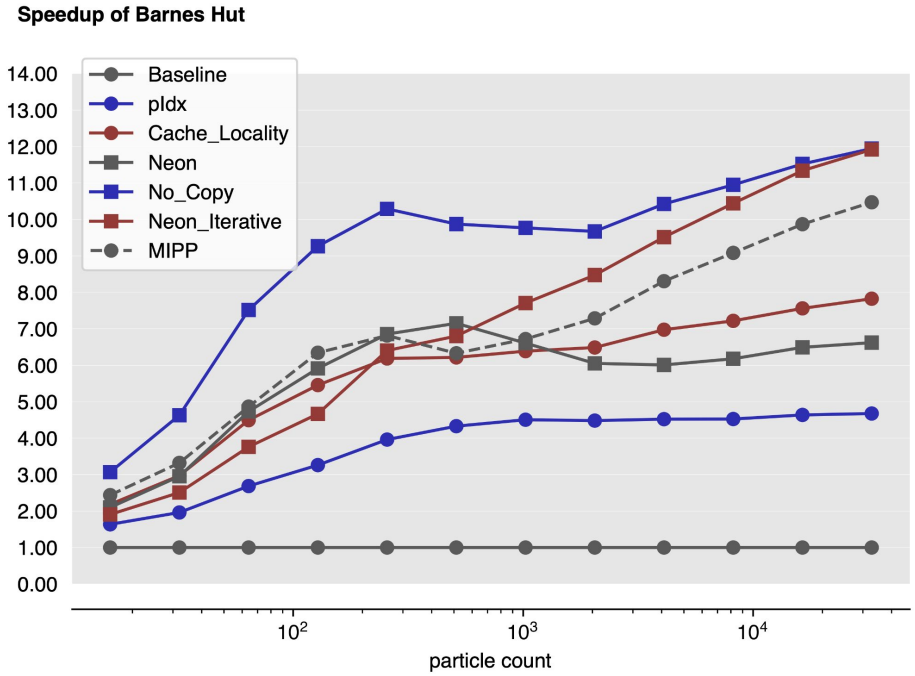


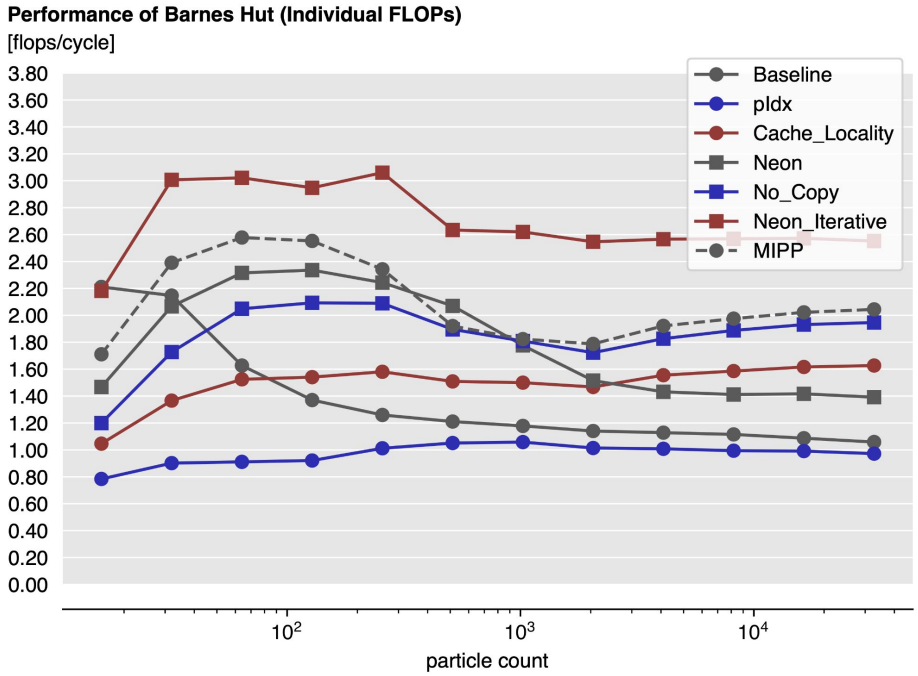
MIPP

- MIPP library (https://github.com/aff3ct/MIPP)
- Platform agnostic variant of SIMD code
- For the NEON backend: Issue in the library, concretely to return false results for square roots (Thanks to @Emanuel for fixing it)
- Performs better than our AVX512 version, which we assume is partly due to optimized mask arithmetic
- For NEON, it however performs slightly worse (unoptimized stores)



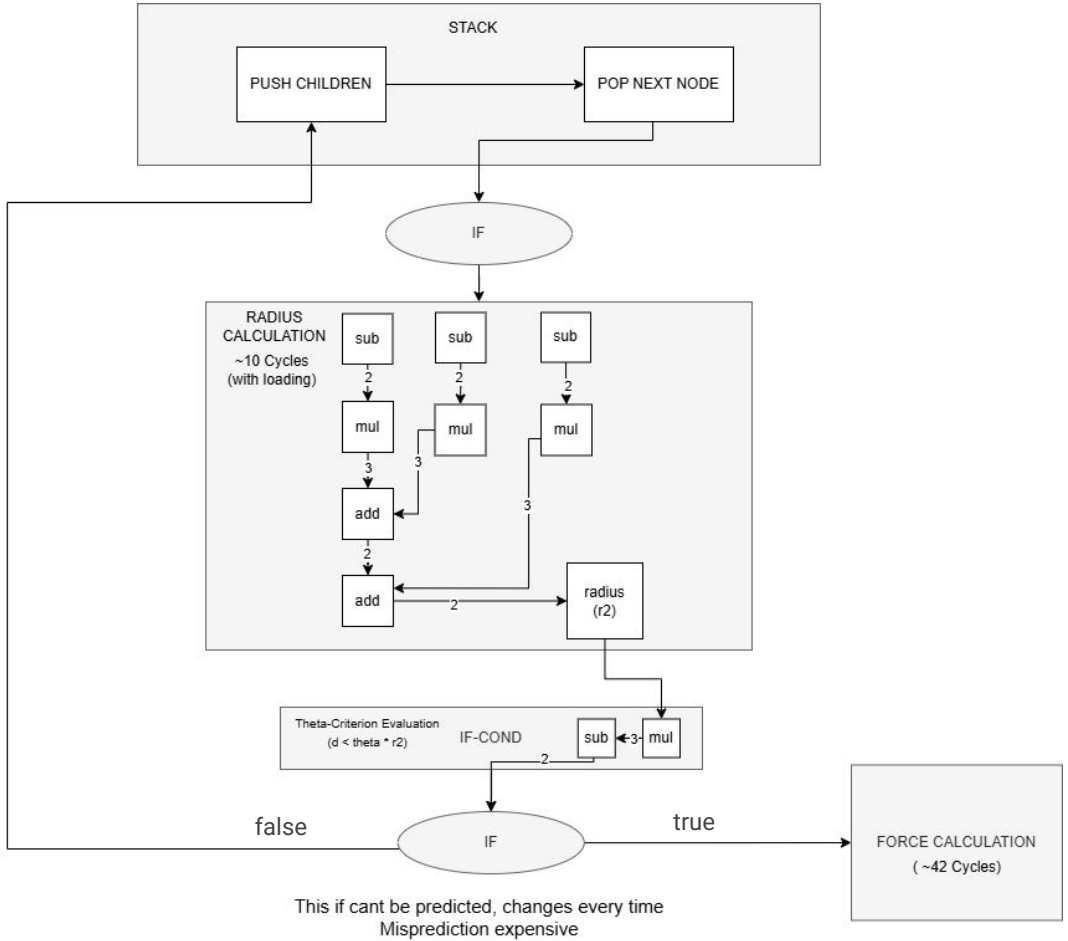
MIPP - Plot


Speedup of Barnes Hut
Baseline
pIdx
Cache_Locality
Neon
No_Copy
Neon_Iterative
MIPP
14.00
13.00
12.00
11.00
10.00
9.00
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
10^2
10^3
10^4
particle count


Performance of Barnes Hut (Individual FLOPs)
[flops/cycle]
Baseline
pIdx
Cache_Locality
Neon
No_Copy
Neon_Iterative
MIPP
3.80
3.60
3.40
3.20
3.00
2.80
2.60
2.40
2.20
2.00
1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
10^2
10^3
10^4
particle count

The ILP Bottleneck

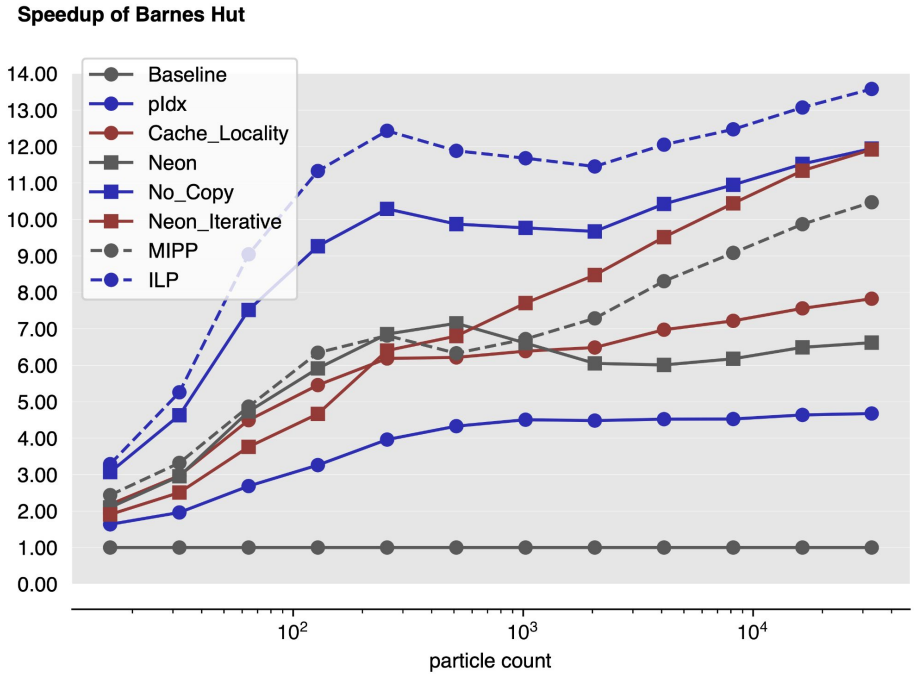
```
while (sp) {
    size_t ft_idx = stack[--sp];
    if (ft->p_idx[ft_idx] != PIDX_EMPTY_OUTER_NODE &&
            ft->p_idx[ft_idx] != self_p_idx) {
        const double other_x = ft->posX[ft_idx];
        const double other_y = ft->posY[ft_idx];
        const double other_z = ft->posZ[ft_idx];
        const double other_m = ft->mass[ft_idx];
        const double side = ft->maxD[ft_idx];
        const double side2 = side * side;
        const double dx = other_x - self_x;
        const double dy = other_y - self_y;
        const double dz = other_z - self_z;
        const double r2 = dx * dx + dy * dy + dz * dz;
        if (side2 < theta2 * r2) { // accept
            // calculate forces
        } else { // open
            stack[sp++] = FT_CHILD(ft, ft_idx, 7);
            ...
            stack[sp++] = FT_CHILD(ft, ft_idx, 0);
        }
    }
}
```

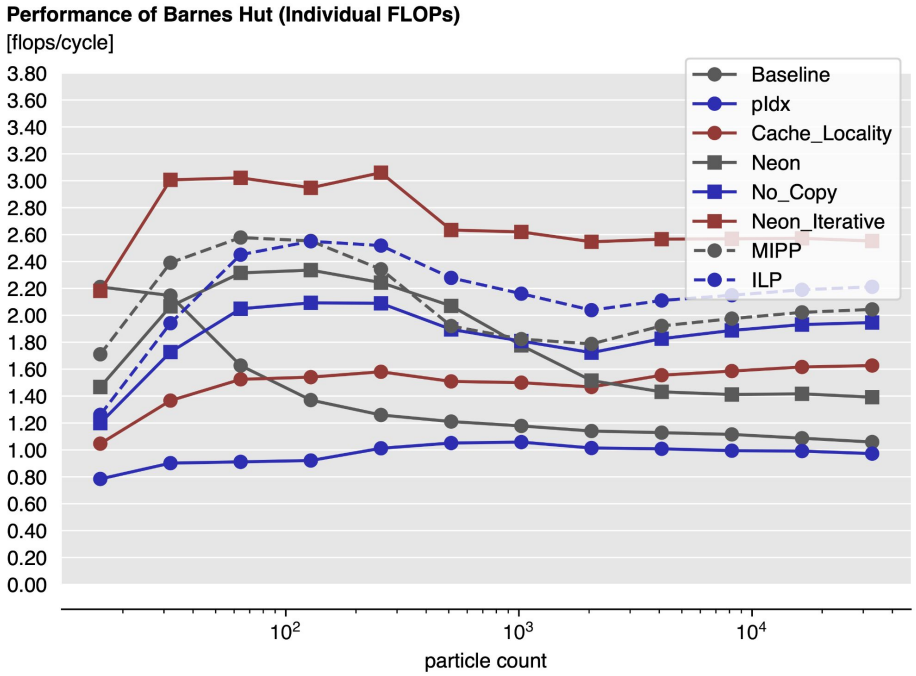


Increasing ILP

- Idea: Increase ILP by “unrolling” the push of children to stack
- Replace the push operation with the entire traversal logic and force calculation
- Leads to speedup that is actually higher than original AVX/NEON implementations
- Implemented using recursive macros (~45000 LOC)
- If “depth” gets too high => Performance loss,
  Likely due to machine code getting too large
- Fastest Scalar version 🎉

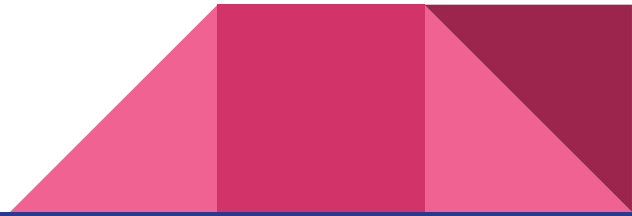
Increasing ILP - Plot



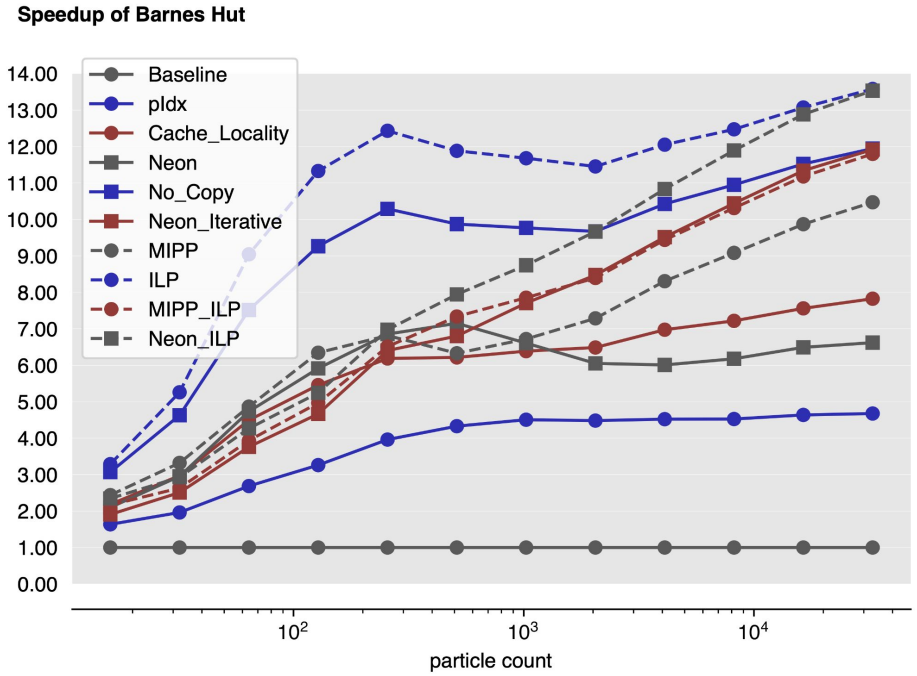


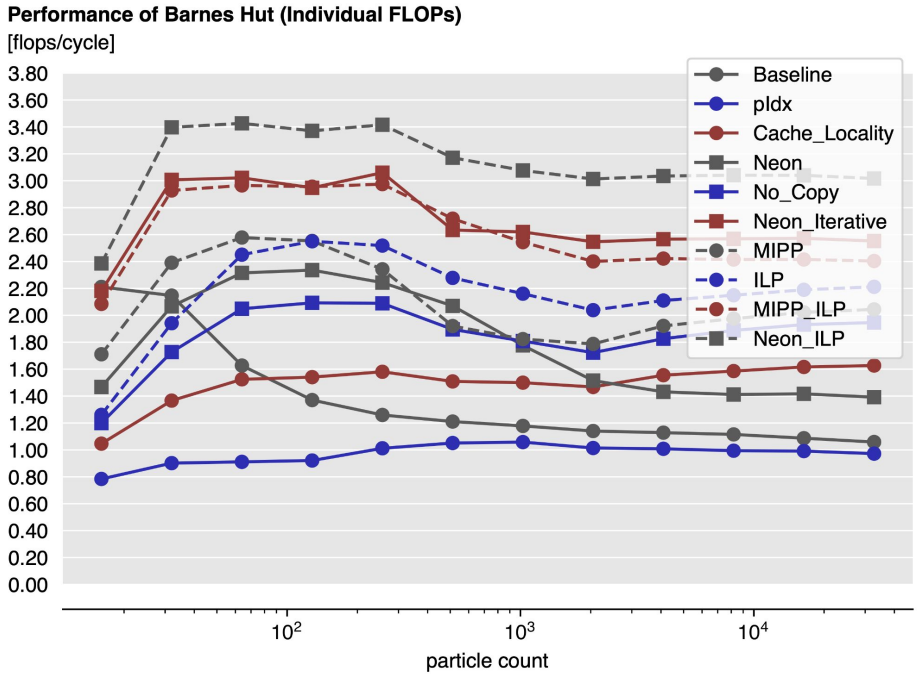
SIMD ILP

- Implemented ILP versions for NEON and MIPP variants
- Manually tuned the inlining level for different machines
- MIPP ILP on AVX512 Server is our fastest implementation
- NEON ILP is implementation with highest performance

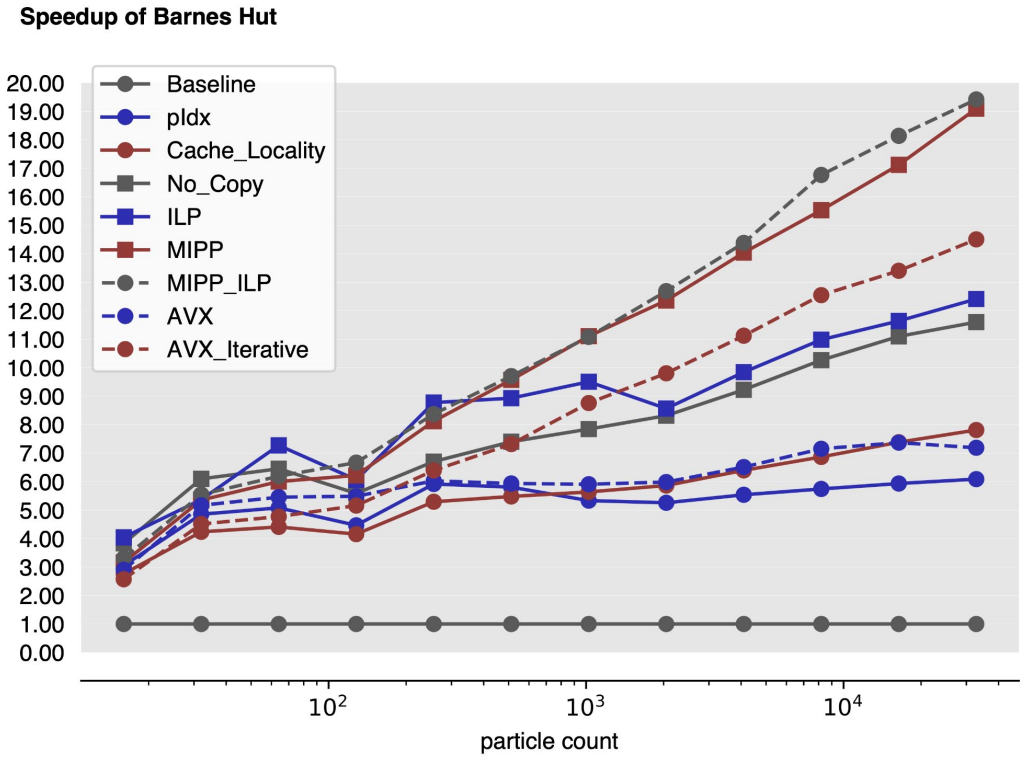


Final Plot

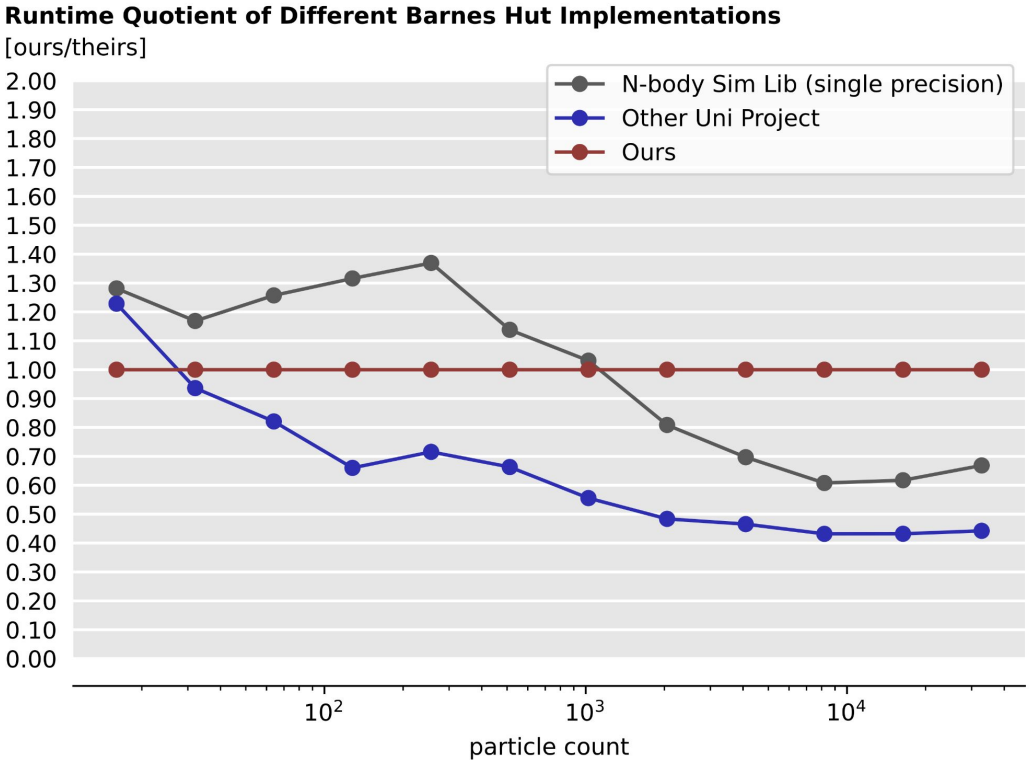

Speedup of Barnes Hut
Baseline
pIdx
Cache_Locality
Neon
No_Copy
Neon_Iterative
MIPP
ILP
MIPP_ILP
Neon_ILP
14.00
13.00
12.00
11.00
10.00
9.00
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
10^2
10^3
10^4
particle count


Performance of Barnes Hut (Individual FLOPs)
[flops/cycle]
Baseline
pIdx
Cache_Locality
Neon
No_Copy
Neon_Iterative
MIPP
ILP
MIPP_ILP
Neon_ILP
3.80
3.60
3.40
3.20
3.00
2.80
2.60
2.40
2.20
2.00
1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
10^2
10^3
10^4
particle count

AVX Speedup


Speedup of Barnes Hut
Baseline
pIdx
Cache_Locality
No_Copy
ILP
MIPP
MIPP_ILP
AVX
AVX_Iterative
20.00
19.00
18.00
17.00
16.00
15.00
14.00
13.00
12.00
11.00
10.00
9.00
8.00
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
10^2
10^3
10^4
particle count

Comparing to alternatives



Thank you!

Q&A?