CGRA Acceleration of the Barnes-Hut Algorithm

Parker Huntington huntingt@mit.edu Quan Nguyen qmn@mit.edu
Daniel Sanchez sanchez@csail.mit.edu

May 2021

Abstract

We observed that implementing tree traversal algorithms on CGRAs was difficult due to the variable amount of state required to complete the tree traversal. Since tree traversals are a wide and important class of problems, generating an implementation framework with additional support for these algorithms is important.

The Barnes-Hut algorithm provides a good starting point for this since it is relatively important in astrophysics and has some nice properties that allow it to be used as a starting point. Starting with previous CGRA work Fifer[5], we modify the underlying Barnes-Hut data structures, and add hardware to the CGRA in order to provide an efficient and reasonable implementation. This implementation is about 6.3x faster in total cycles performance compared to an equivalent multi-threaded CPU version, and with some additional work can be generalized to other more challenging problems.

1 Introduction

Many sorts of "irregular" algorithms present problems for traditional computer architectures. "Irregular" algorithms are ones for which control flow, or data access is inconsistent. Irregular algorithms are very important because they often involve things such as sparse data structures which can be instrumental in decreasing the amount of computation in a problem.

In this paper we will look at using *pipeline parallelism* to improve the speed of the Barnes-Hut algorithm [1]. This algorithm aims to efficiently calculate the gravitational forces on each of a number of point masses. This problem is known as the N-body problem.

In the N-body problem, since the gravitational force from a body affects all of the others, the total number of pairwise force calculations scales as the square of the number of bodies. The Barnes-Hut algorithm improves this by putting each object into an octree, where the nodes of the tree contain the center of mass of their children. An octree is a tree where each node represents a volume split into octants that can each contain another node or a leaf (a body). If a node is small and far away compared to an object, the node's center of mass can be used instead, reducing the number of force computations. This makes the algorithm $O(n \log n)$ instead of the naive $O(n^2)$.

While Barnes-Hut isn't the fastest asymptotic solution to this problem, it is still important and used in large scale supercomputer simulations of things like planetary rings[3] and star cluster dynamics. Thus, the Barnes-Hut algorithm has considerable utility in accelerating beyond its use as a case study in the implementation of irregular tree-traversal algorithms on CGRAs.

We will first give a short background covering the current architectures and their problems, before looking at the architecture that this paper is based on and other related work. Finally, we show that these methods result in significant performance improvements of 6.3x and can be generalized to other related problems.

1.1 Technical Background

Firstly, CPUs are based around the idea that they follow a thread of instructions in order. This paradigm is inherently serial, and thus computers are limited by their ability to quickly execute an instruction in the thread and move onto the next. In the past, Dennard scaling allowed chip fabrication improvements to translate into faster clock speeds, and thus faster per instruction execution. Now that things such as short-channel effects have caused significant frequency increases to become impractical, computer architectures have needed to respond to continue to increase computer performance. The overarching solution to resolve this problem has been to look for *parallelism* that can be exploited to do multiple things at the same time, and thus increase performance.

In CPUs, this has come in three flavors. The first called instruction level parallelism (ILP) is to figure out which instructions can be executed at the same time, and try to resolve dependencies at run-time. Out of Order (OoO) processors use this principle as they can dynamically reorder instruction execution, while still maintaining the original serial ordering in terms of apparent results. Figuring this out, however, is complicated and the number of instructions that can be issued every cycle is limited. The second solution is to split the program into multiple threads of execution, thus giving the CPU more instructions per cycle to execute. This is called thread level parallelism. The final solution is where one instruction acts on multiple pieces of data (SIMD), and is data level parallelism.

It is important to recognize that Barnes-Hut is actually easy in the sense that the force on each body can be calculated independently. The algorithm is embarrassingly parallel. Thus, thread and data parallel implementations are actually quite easy, and the previously mentioned solutions are sufficient, at least at first glance. In reality, the computations are sufficiently simple that the computer quickly becomes bottle necked by its ability to access memory.

In order to understand this bottleneck, it is important to understand cache and cache locality. In computer systems, the working memory often has small fast local copies of the most recently and frequently used data (the cache) that is close to the CPU, etc. If the memory accesses happen in order, then it is easy for the computer to prefetch the memory before it is needed, preventing an expensive access to main memory. When memory access becomes unpredictable, prefetching becomes difficult. Reordering memory accesses can then become very important to make sure that the data is usually in the cache when accessed. That is the case in this paper.

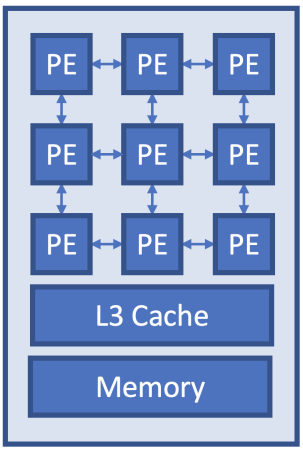


Figure 1: The processing elements (PEs) are tiled across the chip. Each PE is connected to the Last Level (L3) cache which interfaces with the external memory. The PEs are linked together by inter-PE queues.

1.2 Fifer Architecture

Fifer is prior work that is a variation on traditional Coarse Grain Reconfigurable Array (CGRA) architectures [5]. Similar to how a CPU consists of a number of cores tiled across a chip, CGRAs (and Fifer) are composed of Processing Elements (PEs) tiled across the chip (Fig. 1). CGRAs are similar to FPGAs but configured on a much coarser level than LUTs, hence "Coarse Grain".

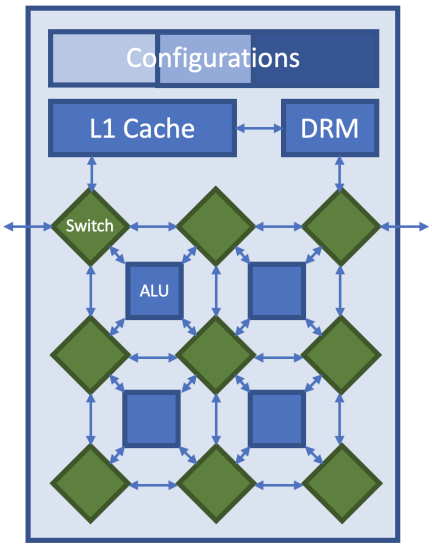


Figure 2: Each PE has a number of different configurations that it can switch between. These configurations set up the switches and ALUs into a pipeline.

1.2.1 Processing Element (PE)

Each processing element is itself another mesh of even smaller elements (Fig. 2). The difference between the processing element and the chip overall is that the PE acts as an atomic unit. All of its elements either process data, or don't during a cycle.

It is made up of a combination of switches that route incoming data to either another switch or an ALU. The ALUs are then configured to do a single math operation once every cycle. These math operations don't change like in a CPU, instead they are set by one of several configurations that is stored in the PE. These configurations don't switch very frequently because all of the data that is in the system during a switch needs to be pulled out and swapped in with the data from the other configuration.

The switching of the configurations and the associated data is what differentiates Fifer from a regular CGRA. Each PE is actually several virtual PEs in essence that can swap out between each other.

This swapping action allows configurations with different amounts of work to load balance the system, and also allows producer configurations to buffer work for consumer configurations on the same PE.

1.2.2 Queues

Queues are the primary means of communication both between different processing elements and within processing elements. Queues are critical because they provide a buffer between processing elements. By allowing a producing stage (or PE) to work ahead, a consuming stage can have a backlog of work that will keep it busy even if the producer stalls for some reason.

Since Fifer and CGRAs in general also focus on a pipeline style of execution, queues are a natural part of the architecture.

1.2.3 Decoupled Reference Machines

Each PE also has some small Decoupled Reference Machines (DRMs) (Fig. 2). A DRM sits between two queues and acts like a very simple stage. This allows enqueued values to be, in the case of Barnes-Hut, treated as indices into an array. When the DRM dequeues each index, it will return the memory value at that index.

By working on the backlog of data in the queue, they can effectively prefetch memory accesses that will be used later.

1.2.4 Scheduling

Since PEs can have multiple configurations, a scheduling policy is needed to determine which one to run. The scheduling system in Fifer is a very simple greedy scheduler. It is greedy in both that it schedules stages with the largest input queues and in the amount of time that a selected configuration runs. A configuration will only be swapped out when it is unable to make further progress.

This scheduling system can run into problems where one configuration "steals" too much time on a PE, but that is not addressed here.

1.3 Why Use Fifer?

The Fifer architecture flips the earlier CPU paradigm of a serial list of instructions by instead having a static set of instructions that define a number of consecutive operations, and data that is serially passed from one instruction to the next. These consecutive instructions are what form the "pipeline" of *pipeline parallelism*. The parallelism comes from the ability to process multiple pieces of data at the same time in different pipeline stages. Compared to other architectures, this one ultimately gives more latitude to reorder the memory accesses and improve cache locality, while still having good computational performance.

1.4 Related Work

Besides Fifer, there has been a lot of work on the Barnes-Hut algorithm. On of the large areas of focus has been on GPU implementations. Unfortunately, due to reasons explained in 3.1 it isn't possible to do a meaningful GPU comparison with our testing methodology. Some of these implementations have focused on being able to execute the full $O(n \log n)$ algorithm including building the tree data structures on the GPU to avoid GPU-CPU bottlenecks[2]. Our approach here is to instead focus on the main computational portion of the algorithm.

Others have focused on improving CPU caching characteristics to lower power consumption[4]. While the cache is very important in this paper, the goal is performance rather than improved power efficiency. Additionally, this paper aims to improve the system around the cache (the execution order and architecture) rather than the cache itself.

2 Methods

We will use the Barnes-Hut algorithm and the Fifer architecture to show how each can be modified to work together and speed up the final result. First, we will show some of the critical adjustments to the algorithms data structures. Then, we will look at the Fifer implementation. Finally, we will see how the Fifer architecture can be augmented to further speed up the calculation.

2.1 Barnes-Hut

When speeding up the Barnes-Hut algorithm, it is first necessary to understand its challenges. Since the forces on each body in Barnes-Hut can be calculated separately, as noted earlier, the problem is embarrassingly parallel. Thus, the obvious solution would be to compute as many of them as possible at the same time.

The problem, however, is that the force calculations are held back by the long memory access times. The memory jumping of the octree traversal causes the cache locality to be poor. These computations need to be intelligently reordered to increase locality. Additionally, tree traversal is traditionally a recursive problem. Each recursion has an associated stack frame which causes the memory foot print to be variable in size (and much larger than necessary). Doing a bunch of tree traversals in parallel would require a large variable amount state data (stacks) that would complicate getting good cache locality. But in a CGRA implementation, the goal is to run multi calculations in parallel in a pipeline fashion. Because the queues holding the state data for the tree traversal computation have a fixed size, the variable amount of state data is a big problem for efficient implementations.

2.1.1 Tree Optimization

The key observation to resolve this problem, is that the crux of the algorithm is to look at a node, and decide whether or not it would be accurate enough to use its center of mass to calculate the force, or if all of the children need to be recursed into. Thus, when a node is visited either all of the children or none of children are accessed, and only data from the current node is needed. In other words, even though there are eight children to a node, either all or none of them are recursed into, and thus the decision is binary.

We can use these restrictions on the octree traversal to construct a "slim octree". All of the nodes are

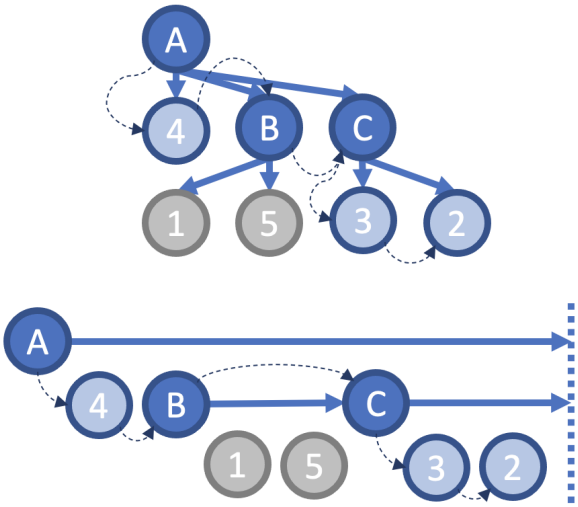


Figure 3: The top shows a standard octree while the bottom a reordered one. Leaves are light blue with the body index, while nodes are dark blue. The arrows on the bottom graph show the skip locations for each node. The dashed arrows show an example traversal for each tree. In this example node B was used in the force calculation, while node C was recursed into in order to provide a more fine grain force calculation. The 1 and 5 leaves are greyed out because the are skipped. The dashed line on the right of the bottom graph is the end of the array that the graph is embedded into. Notice how the bottom graph has a very simple left to right traversal.

sequentially ordered in an array where each node is immediately followed by all of the hierarchy that is under it. The node then contains a single pointer which gives the "skip" value that steps over the hierarchy under that node. An example transformation of this type is give in Fig. 3.

In Fig. 3 node B can be recursed into by incrementing the pointer by one to body 1. If node B is used for a calculation, its children can be skipped by taking the skip pointer to node C. This transformation sequences the accesses (increasing array locations), substantially decreases the node sizes, and allows the tree traversal to be represented by the index of the current node, not an entire stack.

This optimization does have the downside of decreasing the opportunity to compute sibling nodes in parallel (this was possible before because the siblings were all listed in the node), but since we already have plenty of parallelism due to the many bodies, this isn't a problem.

2.1.2 Body Sorting

In order to increase cache locality of parallel calculations, it is helpful to sort the bodies so that adjacent bodies tend to have very similar access patterns [2]. Fortunately, inserting the bodies into the octree has already sorted them to an extent. Bodies that are close together in the tree hierarchy will tend to be close together in space. Thus, we can very cheaply read out the tree in order, and swap the objects to match the ordering (Fig. 4).

2.1.3 Tiling

In a two dimensional representation of the memory accesses where one axis is the node index, and the other the body index, we can imagine some rectangle that encloses some number of computations. This equates to the tiling multiprocessing technique. By tuning the tile size, we can ensure that its associated data will fit into the cache. And thus cache locality is improved.

In a more concrete sense, this can be implemented in a CPU by interrupting the tree traversal at regular checkpoints, and then switching to another object's

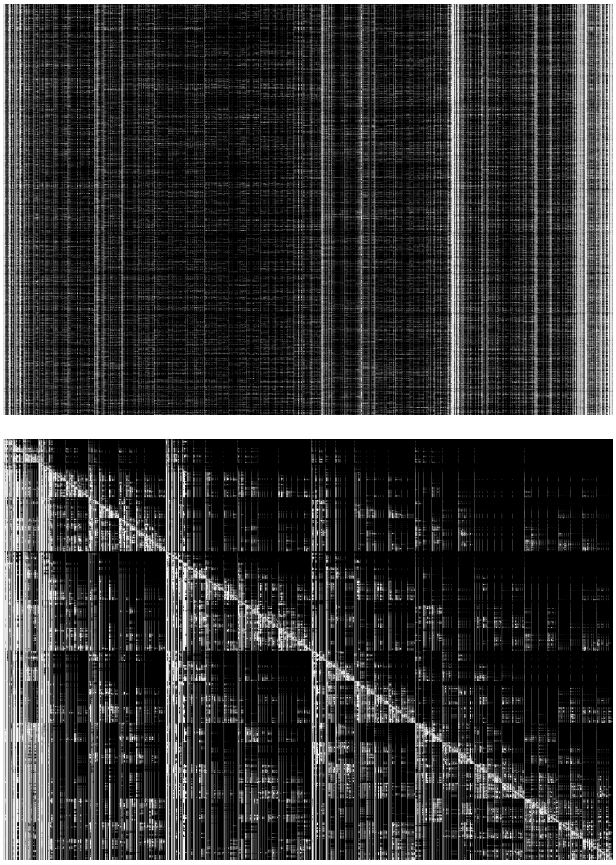


Figure 4: These images show memory accesses in white. The horizontal axis are tree accesses while the vertical access shows the body breakdown. The top image is unprocessed, while the bottom is sorted. Notice how the bottom image has much more structure to it. We can exploit this structure to improve data locality.

tree traversal. Even though this adds a roughly 30% overhead in terms of the number of instructions, if the number of objects being switched between, and the checkpoint intervals are chosen carefully, then the working memory can be optimized to fit into either the L1 or L2 cache.

While this technique isn't directly applicable to the CGRA implementation, the strategy is similar and it is important for providing a fair CPU baseline.

2.2 Combining Fifer and Barnes-Hut

CGRA style architectures can have very large data through-puts since each ALU of a PE can be doing a math operation every single cycle. This means that cache locality for Fifer is still the driving factor for performance as it is in other architectures. In other words, there is still a memory bottleneck.

The first step for implementing Barnes-Hut on Fifer is to look at how the computational state can be represented. Each of the N-bodies has an associated index to identify, as well as the state from the tree-traversal. The tree traversal state is composed of the running gradient (force) calculation, as well as the location in the tree (node index). This can be expressed as something that we will call the *ING* (Index Node Gradient):

```
struct ING {
    uint32_t    index;
    uint32_t    node;
    Vec3        gradient;
};
```

The execution state is then the set of all of the *ING*s that correspond to each of the N bodies.

We can now talk about the action of the compute kernel on a single *ING* element. The first observation is trivially that the *index* field will always remain the same, since one object will never become another. The second observation is that *node* field will either increment by one (continue the pre-order traversal) or it will be set to skip (skip all of the children).

The naive CPU style of executing this would be to reapply the computation operation to an *ING* struct until the computation hits the end of the tree. The CGRA, however, manages a pool of *ING* structs where each go through the computation operation in

a pipeline fashion. These pools are equivalent to the tiling technique presented in 2.1.3. Note, however, that this doesn't change the Barnes-Hut algorithm in any manner. Rather, what it does is define a framework on how to intelligently reorder the computations in the Barnes-Hut algorithm.

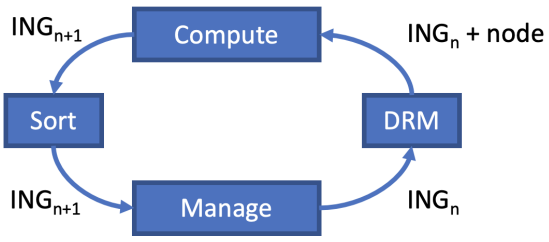
To put this implementation in more concrete terms, in Fifer we create two stages, a computation stage and a management stage. The computation stage is the kernel of the Barnes-Hut algorithm that operates on the pool of *ING* structs (Fig. 5). The management stage simply takes the structs coming out of the compute stage and feeds them back and fetches the node unless the *node* field indicates that the computation for that particular object has reached the end of the tree. Specific to Fifer, these two stages live on the same PE, which re-configures between the two different stages as needed.

Since these two stages are connected to each other by queues, the *ING* structs are never reordered and always processed in a round robin fashion. When a pool of structs starts, each are at *node* 0. Thus, all of the nodes in the pool can reuse the same memory access which enormously reduces the memory bottleneck. Combined with the higher computational throughput of the CGRA, this is what makes the Fifer implementation faster. After node 0, however, the set of node positions in the tree will diverge over time as each object will have different amounts of computations. This leads to less cache locality and causes performance to degrade within the pool.

2.3 Hardware Additions to Fifer

The divergence problem within a pool can be solved by adding hardware to Fifer to schedule (or reorder) computations so that the pool isn't spread out over too large of a range of *node* values at any one point in time. This is accomplished by adding a priority queue (or sorter) that sorts the output of the compute stage for the *ING* with the smallest *node* field within some window.

This sorting hardware needs to be very high throughput (preferably 1 value per cycle), so some of the more complicated priority queue designs aren't applicable in this case. For this paper we settled on




```
compute(ING, node) {
    body = bodies[ING.index];

    if (node is leaf) {
        // if leaf, calculate force
        ING.gradient += force(node, body);
        ING.node++;
    } else if (body far from small node) {
        // if node is used, skip children
        ING.gradient += force(node, body);
        ING.node = node.skip;
    } else {
        // else recurse into node's children
        ING.node++;
    }

    return ING;
}
```

Figure 5: The full flow of the Fifer implementation is shown on top with simplified pseudocode for the compute stage given below. In the upper diagram, an ING struct leaves the mangement stage before going to a DRM that fetches the corresponding tree node. The compute stage then uses the fetched node and the ING struct to do one compute pass, returning the next ING struct. The sorter then schedules these and passes on the ING structs in a different order to the management stage. If the ING struct indicates that a traversal has reached the end of the tree, then it is retired.

Configuration	Cycles	Speedup
simple	15,074,108,773	0.23x
tiled	13,196,453,078	0.26x
4-core 4-thread	3,402,018,023	1.00x
4-PE Fifer	558,817,294	6.09x
4-PE Fifer w/ sort	537,047,738	6.33x

Table 1: Table comparing the performance results from different test configurations for a run size of 262,144 bodies. "4-core 4-thread" is taken to be the baseline since it is the best effort version on the CPU.

a size of 16 entries, which appears reasonable when pipelined into 2 stages.

One of the finer details, however, deals with how Fifer needs to interact with the priority queue. The priority queue is useless if it only has a single entry, but it also needs to output values into the downstream queue so that the management stage can be scheduled. If the priority queue tries to greedily fill itself up, then the system will deadlock when the number of *ING*s has fallen bellow 16 (due to objects finishing the traversal and retiring).

The policy to resolve this problem is as follows: dequeue from the priority queue when there is space in the output, and either the output queue is empty or the priority queue is full. This policy takes care of the edge conditions while also trying to keep the queue full to maximize its effective sorting window.

3 Evaluation

We evaluated the performance of both the Fifer implementation and the CPU implementation using the Fifer simulation software. At a high level, the cycle based simulator is based on and validated for a high performance OoO (Out of Order) CPU. The Fifer simulator uses the ILP in a program combined with the per operation uop requirement to generate a latency value for each stage pass. This technique allows moderately modified x86-64 programs to run as if they had an equivalent Fifer implementation. Thus, the various implementations can share the same code base, and all of the applicable optimizations are shared as much as possible. We also use the exact same data structures between implementations and compare the outputs using a bit-wise comparison for equality.

The test data for the benchmarks is generated using the Plummer model [6]. While this model isn't accurate in terms of star clusters, it is commonly used and should work just fine for comparative results. The randomization seed for each benchmark is also held constant so that the final results are as consistent as possible.

For a system of around 262k bodies, we can look at the performance improvements between each of the optimizations presented (Tab. 1). There are a couple of interesting points to note in the results.

First, the "4-core 4-thread" version is only about 3.88 times faster than the single threaded version even though it has 16 times as many threads. This is due to the fact that having multiple threads per core doesn't help when each core is memory bottlenecked by its associated L1 and L2 caches. Additionally, since the L3 cache serves as the shared LLC in this system, and the working set of the data is is at least twice as large, even though the L3 cache is scaled up for the 4-core version, cache conflicts between different cores can still cause issues. There is also a greater chance for work to not be shared as equally among the 16 threads. This issue is exacerbated by the sorting preprocessing steps described earlier and can be seen in Fig. 4.

Second, while the addition of hardware sorting only adds a minor improvement of around 4-10%, it has a fairly small associated cost. Additionally, there is probably room for significant improvement in the reordering and scheduling hardware. The current sorter has simple policies for determining when to pass values that limit its ability to sort between different passes of the compute kernel. The sorter also becomes saturated in cases where there is only a small population of elements that need to be heavily prioritized over the rest of the pool.

If we recall the design of the Fifer implementation, we have a pool size that can be tuned. This parameter is important because it dictates the amount of queue storage needed within a PE since the entire pool of *ING* structs is stored within the queues.

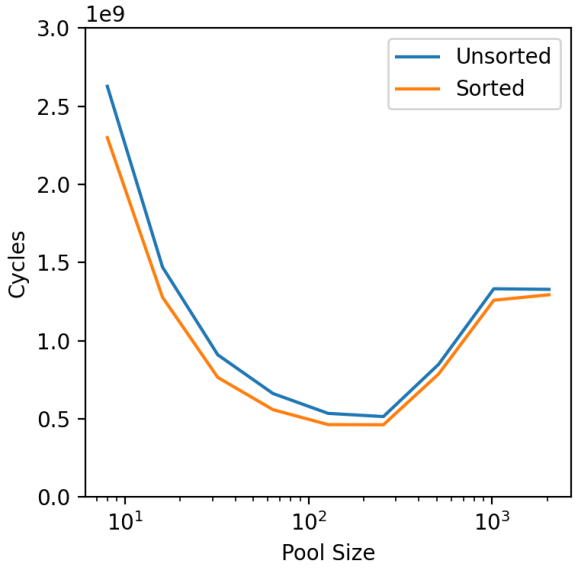


Figure 6: Shown above is the number of cycles vs. the pool size for a single PE fifer implementation with 100k bodies. The optimum point, both with and without hardware sorting, is at a pool size of 256.

Looking at the tuning in Fig. 6, however, also offers some insights into the system. The optimal point here is a pool size of 256. Moving to the left of that value increases the cycle count due to the increased overhead of switching between the compute and management stages as well as the decreased memory access amortization over the pool. Increasing the pool size beyond the optimization point also increases the cycle count due to the pool straining the cache.

Additionally, the sorting hardware and *ING*-style execution can be generalized to some other types of tree traversal algorithms. For example, in a tree traversal that can generate more tree traversals, new *ING*-style structs can be created and added to the execution pool when there is room.

3.1 Limitations

Since the benchmarks were written specifically for this purpose, they have a couple of simplifications that wouldn't be appropriate for actual use. The gradients, for example, are not accumulated in a numerically stable manner (we use a simple running total), although this problem could be solved without too much effort. Additionally, these systems are written using 32-bit floating point numbers. 64-bit numbers would exacerbate the memory bottleneck, however, so the results in this paper should generalize without issue. The benchmarks do calculate the reciprocal square root using full precision, so the computational overhead isn't unfairly low.

The simulation of the CPU style benchmarks does lack hardware prefetching, however, the story here is rather complicated. We can run the CPU benchmark (single thread in this case) natively on the CPU and capture the prefetching behaviour using the hardware performance counters. This reveals that there is a significant amount of prefetching that the hardware is doing. About 70% of all requests to the L2 cache are prefetched, and very few prefetches turn out to be useless. Disabling the hardware prefetchers, however, only affects the total cycle performance to within 0.2%. It appears that the prefetchers probably evict just as much data as they prefetch, and thus have no net effect on the total cycles. Considering the over 200% increase in traffic to the L2 that these generate, their presence is probably detrimental to the power consumption if anything. Thus, we conclude that our CPU baselines are not affected by this factor.

Additionally, the simulation does not implement SIMD instructions. The general observation from the simulation results, however, is that the system is limited by the memory bandwidth which isn't SIMD's strong point in terms of acceleration.

In terms of Fifer, while we have created an operation graph that could be implemented on a CGRA PE, the compute stage of the Barnes-Hut algorithm is rather large (even though Barnes-Hut is relatively simple) so the required PE size may be large or untenable, especially if the compute kernel is made more complex. This problem could be solved by adding

more stages, but that would, of course, slow down the execution due to the need to time multiplex three stages instead of two.

The last major limitation is that the simulator isn't able to simulate GPUs. This is due to the lack of a clear way to map an x86-64 program to the GPU style execution in a sensible and reasonable to use method. While a different simulator could be used, it wouldn't share the same code base to the degree of the current system, and thus would require a significant time investment.

4 Conclusion

In this paper, we showed that CGRAs have the potential to accelerate the Barnes-Hut algorithm. Some of our enabling optimization techniques are also applicable to implementations besides CGRAs. These techniques could allow either faster or more complex simulations of large gravitational systems like planetary rings, to the benefit of astrophysics and other related fields. Our implementation can also serve as a framework to be expanded on for other less well behaving tree traversal algorithms that include complications such as recursively generated tree traversals.

References

[1] Josh Barnes and Piet Hut. "A hierarchical O(N log N) force-calculation algorithm". In: *Nature* 324.6096 (1986), pp. 446–449. DOI: 10.1038/324446a0. URL: https://doi.org/10.1038/324446a0.

[2] Martin Burtscher and Keshav Pingali. "An Efficient CUDA 6 Implementation of the Tree-Based Barnes Hut n-Body Algorithm". In: *GPU Computing Gems Emerald Edition*. MORGAN KAUFMANN PUBLISHER, 2011, pp. 75–92.

[3] Masaki Iwasawa et al. *Implementation and Performance of Barnes-Hut N-body algorithm on Extreme-scale Heterogeneous Many-core Architectures*. 2019. arXiv: 1907.02289 [astro-ph.IM].

[4] Konrad Malkowski, Padma Raghavan, and Mary Jane Irwin. "Toward a Power Efficient Computer Architecture for Barnes-Hut N-Body Simulations". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 146–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188607. URL: https://doi.org/10.1145/1188455.1188607.

[5] Quan Nguyen and Daniel Sanchez. "Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures". In: *UNPUBLISHED* (2021).

[6] H. C. Plummer. "On the Problem of Distribution in Globular Star Clusters: (Plate 8.)" In: *Monthly Notices of the Royal Astronomical Society* 71.5 (Mar. 1911), pp. 460–470. ISSN: 0035-8711. DOI: 10.1093/mnras/71.5.460. eprint: https://academic.oup.com/mnras/article-pdf/71/5/460/2937497/mnras71-0460.pdf. URL: https://doi.org/10.1093/mnras/71.5.460.