FPGA Accelerated N-Body Simulations

Authors: Abhinav Gupta, Rene Ravanan, Yuhe Zheng
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— A system aiming to leverage the efficient programmable hardware of an FPGA to accelerate N-Body simulations. N-body simulations play a pivotal role in astrophysics, molecular dynamics, and a wide range of scientific fields. The report highlights the limitations of traditional CPU and GPU-based approaches, emphasizing the pressing need for more efficient solutions. By leveraging FPGAs, the project aims to achieve substantial speedup and simultaneously reduce power consumption, making it a cost-effective solution.**

***Index Terms*—Acceleration, BRAM, FPGA, N-Body Simulations, Newton's law of Gravitation, Pipelining, Ultra96, Vitis**

1 INTRODUCTION

N-body simulations are fundamental tools in the realm of computational physics, used to model the interactions between multiple particles or bodies within a dynamic system. These simulations play a crucial role in understanding various complex phenomena, including gravitational forces among celestial bodies in astrophysics, the behavior of molecules in molecular dynamics, and even the dynamics of particles in simulations of fluid flow[1]. The primary challenge in N-body simulations lies in the computation of the gravitational forces or other interactions between each pair of particles, which scales quadratically with the number of particles, making it a computationally intensive task.

Traditional approaches to N-body simulations often involve the use of Central Processing Units (CPUs) or Graphics Processing Units (GPUs). While these methods are valuable, they often face limitations in terms of speed and power efficiency, particularly when dealing with simulations that involve a large number of particles that have very irregular computation patterns. To address these limitations, this paper proposes a pragmatic solution: leveraging FPGAs to enhance the performance of N-body simulations.

FPGAs are reconfigurable hardware devices that offer the potential to significantly enhance the performance of N-body simulations while simultaneously addressing power consumption concerns. The driving motivation behind this project stems from the practical need to advance computational physics. Our central goal is to run **2D N-Body simulations** involving a substantial **10,000 particles**, striving to attain a **10x** speedup compared to optimized CPU-based methods.

Our project aims to incorporate FPGA acceleration into computational physics, with a focus on practical applications. This approach is intended to improve computational efficiency, particularly for graduate students and researchers with limited resources. By doing so, we aim to facilitate the simulation of systems involving a substantial number of interacting particles, which is often a costly and resource-intensive task. This practical development seeks to support the scientific community, especially in fields like celestial mechanics and molecular dynamics [1], by offering a more cost-effective solution for conducting research without the need for excessive claims or resource investments. By providing a solution that is more power efficient than its GPU counterparts, we also aim to reduce our negative impact on the environment as well.

2 USE-CASE REQUIREMENTS

In framing our use case for the FPGA-based N Body simulation system, we've established a multifaceted set of objectives that align with our goals. Our primary aim is to achieve a substantial speedup for a large enough simulation size compared to traditional CPU-based simulations, thereby significantly improving computational efficiency and reducing research time. Moreover, it is important that our results are as accurate as we advertise. Not only would it be unethical of us to falsely advertise a product, given that N-Body simulations could have use cases in scenarios like determining if an asteroid is going to collide with the planet or drug discovery in molecular dynamics[1], the onus is on us to ensure that we deliver our promised accuracy as our users could be relying on us in very serious use cases. So, we aim to achieve a **10x speedup for a 10000 particle 2D simulation with an accuracy of 90-95%** (Note that our accuracy requirement is sufficient for our use case[2]). These requirements are explored further in our Design Requirements section.

Incorporating public health, safety, and welfare considerations, our FPGA system will prioritize user safety through robust safety measures. Potential hazards and risks associated with FPGA technology, including overheating concerns, were considered, ensuring a secure research environment. Beyond safety, our approach emphasizes accessibility. The user-friendly design, including a display interface, is developed with sensitivity to diverse backgrounds, promoting a collaborative and globally accessible research environment.

In addition, our commitment extends to environmental sustainability by minimizing power consumption, aligning with environmental goals for responsible high-performance computing. Simultaneously, economic factors play a crucial role in ensuring affordability and accessibility, particularly for students and researchers with limited resources, foster-

ing economic inclusivity within the research community. This approach blends speed, safety, inclusivity, environmental responsibility, and economics to create a comprehensive N Body simulation solution.

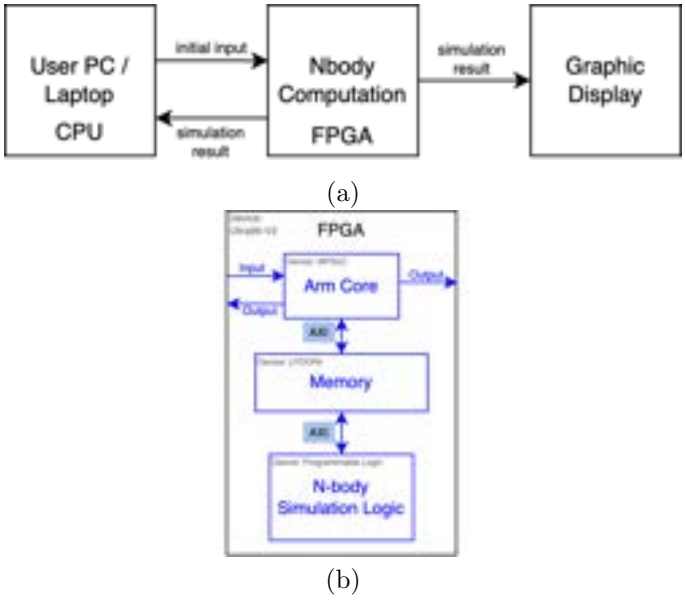

Figure 1: System Description (a) overall system (b) inside FPGA

Figure 2: Data flow diagram describing all pairs algorithm

```
# Main simulation loop
while not simulation_finished:
    # Initialize forces on each particle to zero
    ClearForces()

    # Calculate forces between all pairs of particles
    for i in range(num_particles):
        near_by_particles = []
        for j in range(num_particles):
            if i != j and isNearBy(particle[i], particle[j]):
                near_by_particles.append(particle[j])
        for neighbour in near_by_particles:
            CalculateForceBetween(particle[i], neighbour)
    # Update particle positions and velocities based on forces
    UpdateParticles()

    # Advance simulation time
    UpdateSimulationTime()
```

Figure 3: All pairs algorithm code

3 ARCHITECTURE AND/OR PRINCIPLE OF OPERATION

We implemented our solution with a straightforward architecture. We have a host computer sending the initial simulation conditions (positions of the particles, their mass, and velocity) to our acceleration hardware. The acceleration piece of hardware that we plan to use is the Xilinx UltraScale+ Ultra96v2 development board. This board also has an ARM core integrated into facilitating the running of C++ and a flexible FPGA fabric for accelerated kernels. We plan to have the entire simulation run on the FPGA (using its programmable logic resources including LUTs, Flip Flops, DSPs, and on-chip BRAM) with the results being displayed on a web app where the user can also download them.

Our high-level approach is to have the FPGA's ARM core manage the launching of the simulation kernels, including transferring the particle data to and from the FPGA fabric and FPGA memory. We would transfer data between our host computer and FPGA by securely copying (scp) the files to/from the ARM core. The output files would have a resultant position and velocity of the particles at each time step. Once the specified number of iterations for the simulation has completed, this will be sent back to the user, as mentioned above, and also be sent to a web server hosting a graphical display as a visual reference. Note that the user also has the option of sending data to the web server live during a simulation so they can view their results in real-time. This would help our user gain an intuition of whether or not their simulation is going as expected without having to wait for ours to see their final results. Figure 1 shows a block diagram representing this.

The main algorithm for our simulation can be split into four discrete steps where our optimizations will take place. These steps consist of arithmetic computation such as multiplication, addition, dot products, etc., and will also comprise data transfer between some data structures that store physical information.

We wrote our code in C++ and used the Vitis development platform to synthesize our logic. Doing so allows us to take advantage of the FPGA's performant infrastructure while still maintaining the ease and familiarity of writing code for a CPU. This allowed us to focus primarily on the actual algorithm and optimization of our design instead of logic description details.

Fig. **??** above describes the all-pairs algorithm which we will be parallelizing. As we can see the algorithm runs in $O(N^2)$ because the outermost loop loops through all the particles 1...N, and the inner loop also goes through all the particles and determines which particles are nearby to the particles $i \in 1...N$. This is after all the nearby particles have been determined we then calculate the force on particle i and then update the position of the particle. After this is done for all the particles 1...N then we increase our time by 1.

4 DESIGN REQUIREMENTS

4.1 Speedup

The 10x speedup for 10000 particles is achieved by optimizing the computational aspects of the simulation. This is our most important requirement as this is the speedup our physicists need in order to make our product viable for them. The motivation behind our speedup goal was to see what was theoretically the maximum speedup we could achieve on an FPGA. The fabric clock on an FPGA runs at around 200MHz[3] which is about 15x slower than an i7-9700 3.0GHz[4] (this is the CPU we are running our reference calculations on). But we have seen that the Cache/-DRAM on the FPGA is significantly faster than the CPU. The memory is somewhere near 10x-100x faster. If we take both of these factors into consideration then we can see a theoretically viable and achievable speedup is around **10x**.

4.2 Simulation Scale

The system should support simulations with a minimum of 10,000 particles in a 2D environment. The scale should be suitable for molecular and astronomical simulations, ensuring meaningful results[5].

This also perfectly fits our hardware resources, on our FPGA we have 70K LUTS which should be able to support data transfer and calculation of around 10000 particles [3]. This number does remain flexible though in case further testing reveals to us that we might be more hardware-bound.

We are very happy with our achieved speedup of 40x because it will be able to cut down the simulation time for a 10k particle simulation that might have taken 20 hours, for example, to merely half an hour. This speedup is crucial in stratifying our use case which is first and foremost being able to help the under-resourced physicists out there. This also helps us achieve our other environmental and collaborative goals that to achieve this 40x speedup as only an FPGA will be needed and no extra components, this will help cut down costs and global waste and will encourage more cross-platform partnerships within academia.

4.3 Accuracy

From our research into the use cases of the N-Body simulations, we foresee a **90-95%** accuracy in our results being sufficient[2]. It is important to define this metric as various implementations of this algorithm tend to lose precision over time. This was also particularly relevant when we were working with fixed point types leading us to choose between the tradeoff of precision vs speedup.

5 DESIGN TRADE STUDIES

For our project, we had to make several considerations and decisions when approaching our problem: Accelerating N-Body simulations in an efficient and cost-effective manner. We first had to choose our acceleration platform, our FPGA, then our simulation algorithm and finally our hardware acceleration approaches.

5.1 Hardware Platform

As mentioned above and in our introduction, versions of the N-Body simulation already exist on CPUs, GPUs and other platforms. When choosing our platform we had to carefully evaluate the costs and benefits of each of them, how they can cope with our workload, their efficiency and accessibility. The four main platforms we considered were a CPU, GPU, ASICs, and FPGA. Below are the pros and cons of each that we considered.

5.1.1 CPU with Multithreading

A comparatively straightforward approach to this problem would be to use a CPU to conduct our simulation. Two of us have already tried this approach in the course 15-418: Parallel Computer Architecture and Programming. The intuitive way to parallelize this simulation is to do so on the axis of each particle during the compute force section of our simulation. This would not be the most efficient on a CPU given that we are dealing with around ten thousand particles at a time and most standard CPUs allow 8-16 threads (with supercomputers using 128). Additionally, thread synchronization and communication costs would be very high given that we would have many shared data structures. Additionally, CPU kernels would also spend time context switching between different applications and processes that are running devoting less time to our simulations and making running them all the more expensive.

5.1.2 GPU

GPUs solve the above mentioned problem of CPUs being hardware bound by the available concurrency on the platform as each block on a GPU can spawn up to 1024 threads making this very scalable for our use case. However, our computation can be very irregular, for example, particles that are grouped together would have to perform larger resultant force computations as compared to sparsely located ones. This would again lead to several overhead communication costs between threads and GPU blocks which significantly hurts our performance. Moreover, the memory architecture of a GPU would force us to

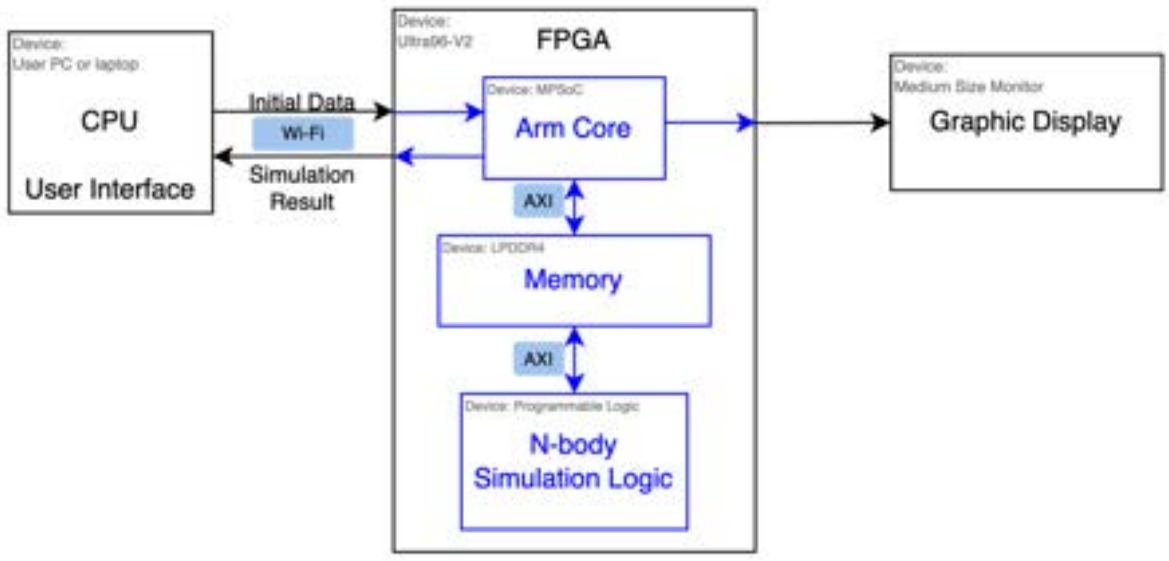

Figure 4: Integrated Block Diagram

use its global memory instead of its shared block memory given that all blocks need to have access to the particle information. Reading/Writing data in the global memory of a GPU is also very expensive. Lastly, a GPU is not very power efficient, an average GPU has a TDP of 300W[6]. Even though some fast implementations of our problem have been designed on GPUs, their adverse effects on the environment with their power consumption make us want to work on a more sustainable platform.

5.1.3 ASIC

ASIC is an efficient approach, however, given the time constraints that we are presented with, we would be able to design and fabricate a chip. Our design would also have to be produced at scale for this to be worthwhile, making this not ideal for a prototype system.

5.1.4 Why an FPGA?

Given the above constraints with other platforms, we chose the FPGA as our target platform. FPGAs allow us to take advantage of customizable hardware to exploit parallelism in conducting our N-Body Simulation. FPGAs also offer high flexibility in their memory architecture with BRAM allowing us to take advantage of memory reuse without the costly high latency trips of DRAM. We have also found well-documented work available for accelerating code on FPGAs including papers[7] that have tried our N-Body simulations on cloud FPGAs. FPGA kernels would also not have to deal with as much overhead as a CPU kernel as the former would be focusing solely on our computation. Lastly, compared to options like ASIC they are also relatively cheaper and is also more power efficient than GPUs (our FPGA has a TDP of only 24W[3]) as mentioned in our design requirements.

5.2 Hardware Acceleration Approach

5.2.1 HLS vs. HDL

After choosing the FPGA as our platform, we needed to decide how we were going to write our kernel code. This could have either been done using a Hardware Description Language (HDL) like SystemVerilog, or using a more programmer-friendly language like C++ in conjunction with High-Level Synthesis. The first consideration we had was that our initial CPU implementation was already written in C++, so choosing C++ with HLS would mean that we could spend more time optimizing a familiar and approachable algorithm instead of spending time getting an initial implementation setup and running. Additionally, HLS also offers several tools to facilitate parallelism, such as pipelining and unrolling Pragmas on Vitis HLS, and the compiler also is able to infer and parallelize independent sections of our code which programming in SystemVerilog would not allow.

5.2.2 Fixed-Point vs Floating Point Numbers

In our initial design, we did intend to use fixed point type numbers instead of floating points as they would take up less hardware. However, after implementing a version of our code with this data type, we found that we were able to gain a significant speedup but this was not entirely accurate. Given that several steps of our computation include really large/small numbers like the gravitational constant G, we found that our fixed point version was zeroing out a majority of our compute force computation leading to us

falsely achieving a speedup. For this reason, we pivoted back to using single precision floating point types which offer an actual accuracy metric

5.3 N-Body Simulation Algorithms

After finalizing our hardware platform and acceleration approach, we had to choose the algorithm that we would run our simulations with. There are several proposed algorithms to run N-Body simulations, the two most widely accepted ones are the Barne's Hut and All Pairs algorithms. Their tradeoffs have been discussed below

5.3.1 Barne's Hut Algorithm

The Barne's Hut algorithm[8] has an $O(NLogN)$ time complexity where N is the number of particles. Here at each time step, a particle can find its nearby particles to compute its resultant force and velocity vectors using a data structure called a quad tree. This data structure, as seen in the figure 5, splits the simulation space into quadrants where each node represents one quadrant and continues to recursively do so until each leaf has only 1 particle or has child nodes that split it further. Traversing this to find nearby particles is considerably more efficient ($O(LogN)$ vs $O(N)$)[9]. However, implementing this in HLS would be quite complex and reading/updating this data structure would be quite expensive on an FPGA.

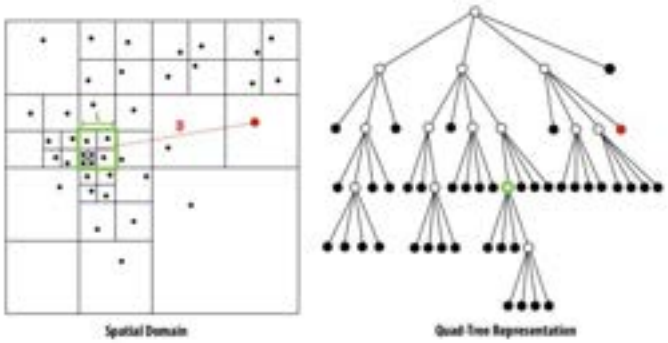

Figure 5: Barne's Algorithm with Quad Trees[9]

5.3.2 All Pairs Approach

This approach is slightly less sophisticated and is the fundamental approach when performing the N-body simulation, at each time step, each particle loops over every other particle to find out if it is nearby and then compute its interactive forces. This approach has a higher time complexity of $O(N^2)$. However, this is a lot easier to parallelize given its comparative independence in force computations and update phases. It is also easier to store into memory on an FPGA making this a more widely used algorithm in accelerating N-Body simulations in platforms such as FPGA and GPU. Hence, given its comparative ease to optimize, we chose the All Pairs approach.

6 SYSTEM IMPLEMENTATION

As mentioned in the architecture section, our hardware platform will be the Xilinx Ultra96-V2 development board. A user can interface with this board to send/receive data from it via WIFI, by copying input and result files to/from the board via scp and ssh. The graphical visualization tool will be hosted on a web server that will receive live data from the FPGA.

With respect to our actual optimizations, we have mentioned our plan below.

6.1 Loop Unrolling

Loop unrolling is a critical optimization technique employed in parallel computing to improve the execution speed of iterative loops within algorithms. In the N-Body simulation, loop unrolling involves unwrapping a loop, essentially transforming a series of repetitive iterations into a sequence of independent and parallelized operations. Instead of processing a single element or body per iteration, loop unrolling allows for the simultaneous handling of multiple elements within a single iteration, making more efficient use of the hardware's capabilities and increasing computational throughput.

The "all pairs" algorithm has nested loops for pairwise interactions. By unrolling these loops, we can effectively perform multiple pairwise interactions within a single iteration, reducing the overhead of loop control and enabling the utilization of vectorized instructions or parallel hardware. This optimization technique aims to significantly accelerate the simulation by decreasing the number of loop iterations and improving the overall runtime efficiency.

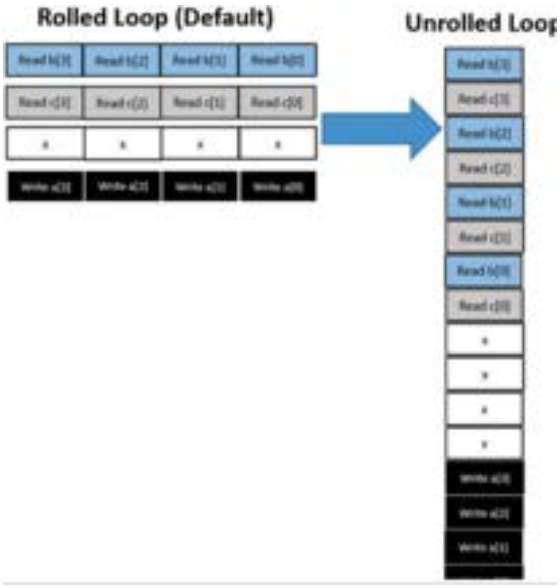

Figure 6: Rolled vs Unrolled loop [10]

6.2 Pipelining

Pipelining is another optimization technique that aims to reduce loop latency. As seen in the figure 7, it does so by starting the next iteration of a loop before the current one finishes. This is different from unrolling as it's possible that different loop iterations may have some data dependencies or just cannot be run concurrently, so pipelining facilitates running loop iterations concurrently while not having these critical sections overlap.

In the N-body simulation, each body interacts with every other body to calculate gravitational forces or other interactions. With the pipelining pragma, these interactions can be divided into stages, such as data fetching, computation, and result storage. The pragma enables overlapping the execution of different stages, making more efficient use of hardware resources.

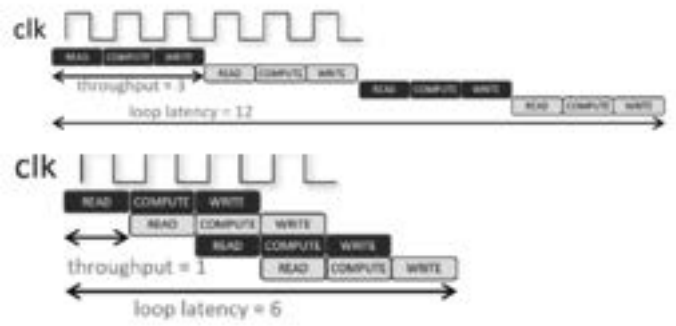

Figure 7: Unpipelined vs Pipelined [11]

6.3 Software Refactoring

The optimizations above alone were not enough to achieve our speedup goals. An important part of our solution was to refactor sections of our algorithm to make it more parallelizable and therefore compatible with the unrolling and pipelining. The two main approaches we implemented are discussed below.

6.3.1 Matrix-Vector Multiplication with Loop Reordering

An important code change to our solution was to separate our actual computation/math from the rest of our code and to treat this as a dense graph. This leads us to mapping our computational model to the matrix-vector multiply model as seen in the figure 8. Our first iteration of this can be seen in the left half of this figure where we have an element-wise compute force where we accumulate each particle force across multiple iterations. This however creates a data dependency in the inner loop leading to pipeline stalls that affected our speedup.

This led to our next refactor where we inverted the order of our two loops. By doing so, as seen in the right half of the image below, we still retained our element-wise force computation but instead of accumulating a resultant force for each particle, we element-wise add forces across particles making our loops independent of each either and easy to pipeline, boosting our speedup.

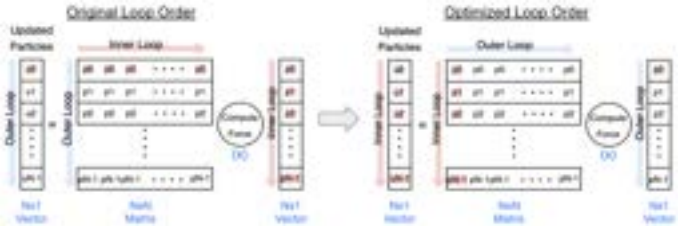

Figure 8: Matrix Multiply Model

6.3.2 Batch Computation

To augment the newly refactored code above, we also grouped our computation in batches to take more advantage of loop unrolling so we could execute a larger portion of our code concurrently. A visual of how this works can be seen below. This coupled with BRAM partitioning greatly facilitated our concurrent memory accesses. With some testing, we found that increasing our batch size led to a higher speedup as seen in the graph below and we found that a batch size of 8 gave us the best speedup (40x!). We could not go beyond this due to the board's hardware resource limitations.

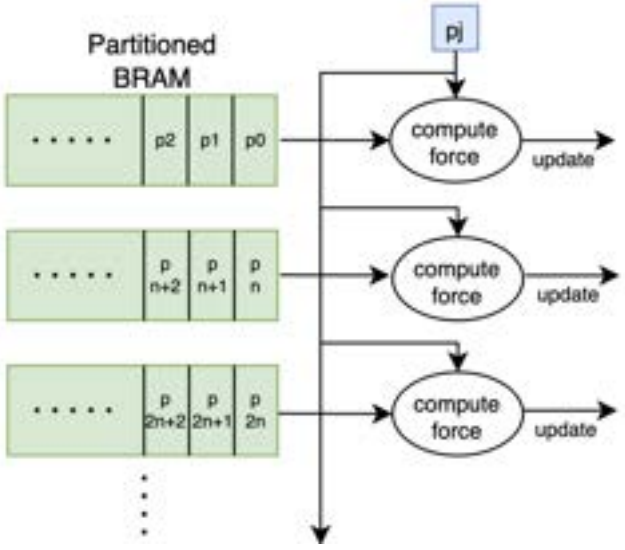

Figure 9: Batch Computation

Figure 10: Batch Size vs Speedup

6.4 FPGA Memory

A major advantage of computation on FPGA is that we can implement custom memory structures to fit the algorithm's data accessing pattern. This way, we can improve the memory throughput which usually bottlenecks memory-intensive computations.

6.4.1 DRAM

The DRAM on the FPGA stores the initialization and simulation results of our system. The host program in the Arm core stores the initial state of the particles received from the user's PC or laptop to the DRAM before the simulation starts. During simulation, the N-Body computation hardware kernel in the programmable logic continuously reads the particles' information - position, velocity, and mass - from DRAM. The kernel calculates the particles and then writes back the simulated results. At the end of the simulation, the FPGA core extracts the simulated results into files for the user to retrieve from the board. The DRAM of the Ultra96-V2 FPGA contains 2GB memory and has a measured read latency of around 500ns - several dozens of cycles depending on the computation's target frequency[3]. This long latency means we need to perform our computation on locally buffered data instead of making frequent trips to DRAM. We also need to batch-read data from DRAM to local buffers to maximize DRAM throughput as we have limited set of DRAM ports (512 bytes total [3]) limiting the degree of our concurrent read/write access patterns.

6.4.2 LUTRAM

LUTRAM is on-chip memory implemented from LUTs in the programmable fabric. LUTRAM can have 32-bit or 64-bit wide ports, and it supports dual-edge clocking for fast data access [12]. Therefore, LUTRAM is ideal for fast and flexible data access. However, because LUTRAM is implemented using LUTs, more data stored in LUTRAM means less LUTs can be used for calculation. Therefore, we should only store data structures that need to be accessed in every cycle in LUTRAM to optimize resource usage.

6.4.3 Block RAM

Block RAM is a hardened fast memory structure in the programmable fabric. Block RAM supports dual port access, which means we can read and write from Block RAM at the same time. Block RAM ports are reconfigurable. It supports reshaping and partitioning memory. This means we can access multiple elements of an array at the same time, enabling us to parallelize a computation by pipelining and unrolling loops. We found that our board has about 432 KB of BRAM [3], which allowed us to store our entire particle data set (about 200KB plus some more for temporary buffers as seen in our resource util below) onto it greatly improving our memory latency.

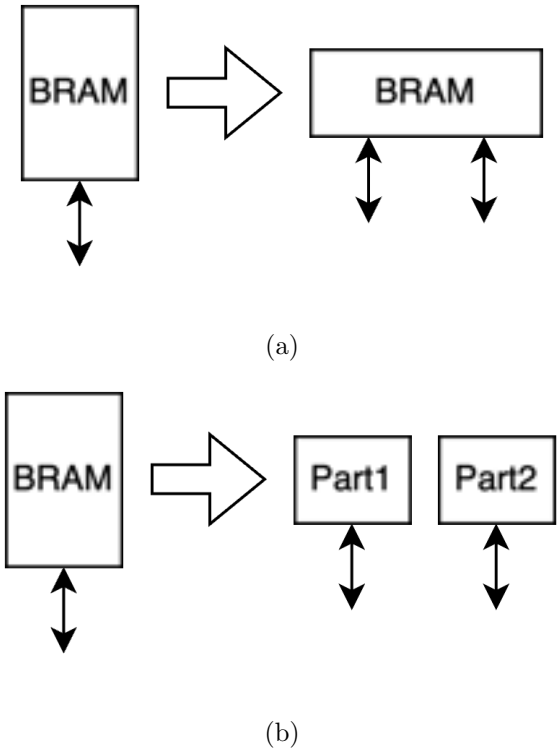

Figure 11: BRAM Configuration (a) Array Reshape (b) Array Partition

6.5 Graphical Display

One major change from our design to our final project was our graphical display. While we initially intended to drive the graphical display on a monitor through the FPGA's miniDP port, we found that working with the graphics libraries on the board to be very cumbersome and we also realised it is wise to separate graphical processing from our FPGA onto a separate Flask server. It is also much better suited for our use case requirement wherein the Physicist can look at the simulation data and the go and that the graphical animation is being run in real time.

After all of these considerations, we decided to finally deploy a lightweight webapp on an online hosting platform.

6.5.1 Choice of web app framework

We choose Flask to build a web application over other web frameworks based on a multitude of factors. In our case, we wanted a lightweight and easy-to-use website that displayed animations and allowed us to easily download files. Flask is simple and lightweight, making it an ideal choice for small to medium-sized web applications. The flexibility enables the easy integration of third-party libraries and tools, which was crucial for us as we needed to integrate other Python libraries like Matplotlib, pandas, and NumPy in order for our graphical simulation to work properly.

In contrast, other popular frameworks like Django offer more built-in features but at the cost of increased complexity and a steeper learning curve. We thought that Django might be overkill for smaller webapp like ours. Ruby on Rails also offers many built-in tools but enforces convention over configuration, which can limit flexibility which for us again was not feasible. Flask's extensible nature allowed us to add only what we needed, ensuring a lean and efficient application. [13].

Lastly, our decision to stick with a Python-based framework was also made because our graphical simulation algorithm was written in Python and so the integration of the web app with the simulation was very seamless.

6.5.2 Choice Of Hosting Platform

Choosing to deploy a Flask web application on Amazon Web Services (AWS) over other hosting platforms was a decision influenced by several key factors. Firstly, AWS offers a lot of scalability options, which are essential for handling our website because depending on the simulation the user is running they might need more CPU (to do the file reading, processing and also animation creation) power and storage just to simply contain (exact of our simulation result file is around 400KB, if the user ran it for even 1000 iterations the file would take up approximately 400MB). AWS also provides a comprehensive set of services like Amazon RDS for database management, AWS Lambda for server less computing, and Amazon S3 for storage, which integrate seamlessly with Flask, allowing for a more streamlined development and deployment process.

Comparatively, other platforms like Google Cloud Platform (GCP) and Microsoft Azure also offer similar services, but AWS stands out for its mature ecosystem and extensive documentation, making it easier for us to find solutions to potential issues.[14] Additionally, AWS's pay-as-you-go pricing model can be more cost-effective, especially for applications with fluctuating usage patterns (the website may lay dormant for a couple of days and be used excessively on other days depending on when the user deploys their simulation). This economic efficiency, combined with AWS's proven track record and widespread adoption in the industry, ultimately guided the decision towards AWS for hosting the Flask web application.

6.5.3 Functionalities on the web app

For our web app we decided that easy of use and access was our priority so we planned on keeping the UI clean and minimal. Our core functionalities are displaying a real time graphical simulation. We have 3 buttons which direct the users to 1) Downloading a ZIP file containing simulation data from past simulation 2) Downloading simulation files from the current simulation (this is more fine grained as the user can specifically decide which iteration's output file they want to download) 3) Download the animation of the current or past simulation. This functionality lets the user download the most up to date animation of the current simulation (Same as the one displayed on the home page) or of a past simulation.An image of how our homepage looks can be seen below.

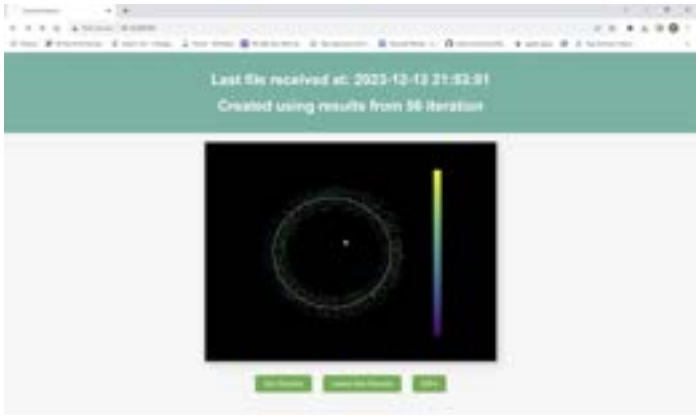

Figure 12: Web Server

6.5.4 Tradeoffs

Some of the trade-offs we made for our web app were that they ran our application on a t2.medium AWS EC2 instance which is a paid instance and costs $0.0464[15]. We did this rather than running it on a free instance because our CPU utilization on the free instance made was hovering around 70% which was too high and the website could not be accessed due to extremely high load, so we decided to switch to paid instance. Even with our paid if the server is constantly up and running for around 7 days straight they would have to pay around $8 which is very reasonable (User can stop the instance when not running a simulation and would not have to incur a cost).

The other major trade-off we made was adding the functionalities of downloading past results, this made the web app a little bit less lightweight and the server would have to serve more content and do more processing when a download request occurs (Probably another reason as to why we had to upgrade our server). But again we thought that this was something we could not leave out as it is very possible the user wants to look at past simulations and not just

the current or that they might want to download a ZIP file containing the whole simulation's data rather than the data from individual time step.

7 TESTING & VALIDATION

7.1 Accuracy Tests

For our given use cases of Molecular and Astronomical N-Body Simulations, we found that a $90 - 95\%$ accuracy would be sufficient[2]. Hence, the first thing that we will check with our project is whether the N-Body simulation results that we produce satisfy the requirements in accuracy. To do so, we have our benchmark CPU implementation which we developed in a previous class as a method of validation. Note that to make this a fair comparison, especially since we switched back to floating point types on the FPGA, the CPU implementation was modified to use double-precision floating point numbers to best evaluate any changes in accuracy.

To get a numerical value for our accuracy we used the formula that intuitively makes sense, we simply take the average difference over all 10000 values over all our iterations between the (x, y) coordinates of the reference solution and our own.

Note - Even though we used we used double precision floating point value for our reference (this serves well for our purpose of comparing our simulation to a model that is more accurate than us.) the CPU implementation is not fully accurate as it is impossible to fully contain position data even in 64-bit float, but again we thought it was good enough to use as a reference value and we since

$$\frac{\sum_{j=1}^{\text{Num Iter}} \sum_{i=1}^{10000} \left[\frac{\left|\frac{x_{ref}-x_{test}}{x_{ref}}\right| + \left|\frac{y_{ref}-y_{test}}{y_{ref}}\right|}{2} \right] \times 100}{\text{Num Iter} \times 10000}$$

Figure 13: Accuracy Verification Formula

Below we have included our results for the same. Note that we mentioned we are comparing our FPGA to a double precision floating point implementation on the CPU. We did also try single precision floating points on both and found that they were always almost a 100% accurate and therefore did not think that this was a fair comparison as we want to compare the best CPU implementation with our FPGA. In the figure 14 we can see that our accuracy scales quite well with the number of iterations which suits our use case. At higher iterations, our accuracy declines a lot more gradually. This is because for our simulations it is impossible to achieve perfect accuracy since all data types would start to lose precision at some point as seen in the CPU implementation also beginning to lose precision at higher iterations slowing down the decline in accuracy of our FPGA keeping it above 90%.

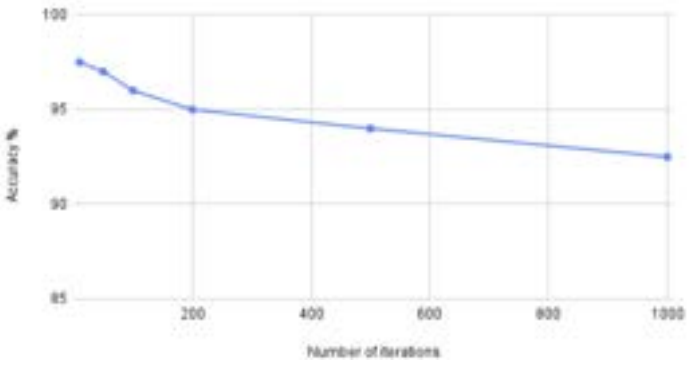

Figure 14: Accuracy vs Number of Iterations

7.2 Verifying Speedup

To measure and verify our speedup we measured the end-to-end run time of our CPU implementation and divided this by the end-to-end run time of our FPGA implementation. The figure 15 below shows the time breakdown of the distribution of our speedup. As mentioned in our design, we aimed to first optimize our data access latency by using BRAM for the initializing and return of particle data but the bulk of our speedup was achieved in our computation section. Note that these timings were taken for a 10 timestep/iteration simulation and averaged to get the per iteration speedup. These results were also averaged across multiple runs of the same input to average out any non-deterministic DRAM timing. These results are also tested against a variety of input patterns to make sure our speedup was not specific to any use case.

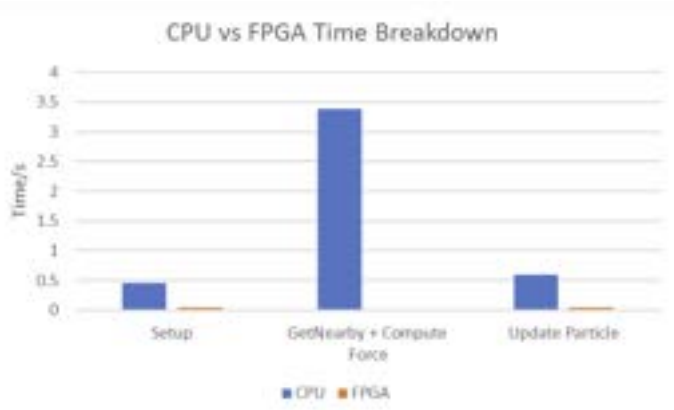

Figure 15: Timing breakdown of Runtime for 1 iteration for 10K particles for 10 iterations

Another metric that was important to explore was how our speedup scales with the number of iterations. It was important that we at least maintained our achieved speedup. The graph below shows the results for the same. Here we can see that our speedup scales up as the number of iterations increases, reaching about 50x for 1000 iterations! This can be explained with Amdahl's law[16]. Given that our bottle was reading and writing the initial and final particle

data to and from BRAM (the sequential non-parallelisable part of our code), increasing the number of iterations leads to us spending more time on our computation therefore increasing the arithmetic intensity of our program leading to our higher speedup. A figure plotting these results can be seen below.

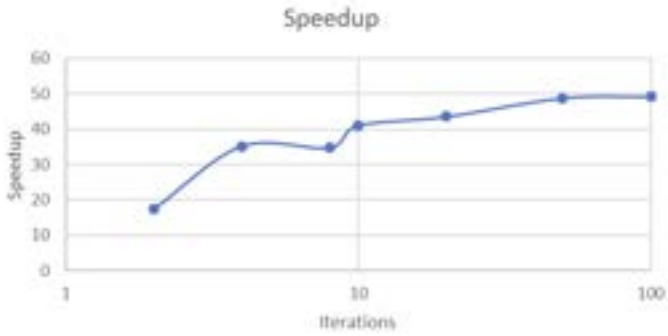

Figure 16: Scaling of Speedup with Iterations

7.3 Power Consumption

Given that we are trying to also be energy efficient, given our achieved speedup of 40x and the Ultra96's 24W TDP [3], we have achieved our goal of being more efficient than CPUs (100W TDP) and GPUs (300W TDP) [6].

7.4 Resource Utilisation

To confirm that we pushed our board to achieve the best possible speedup, we analyzed our Vitis Resource Utilisation reports for the different resources on board (Block RAM, Look Up Tables, Digital Signal Processors, and Computational Logic Blocks). In the graph below, we see that for a batch size of 8, we nearly max out our CLBs hinting that this was the highest speedup we could achieve with our current implementation. Another confirmation of this was that trying to build an implementation with a higher batch size would not succeed. Another thing to note here would be the evident tradeoff between achieved speedup and resource utilization, getting a higher speedup comes at the cost of using up more resources on the board, reaching our 40x speedup led to us maxing out the same.

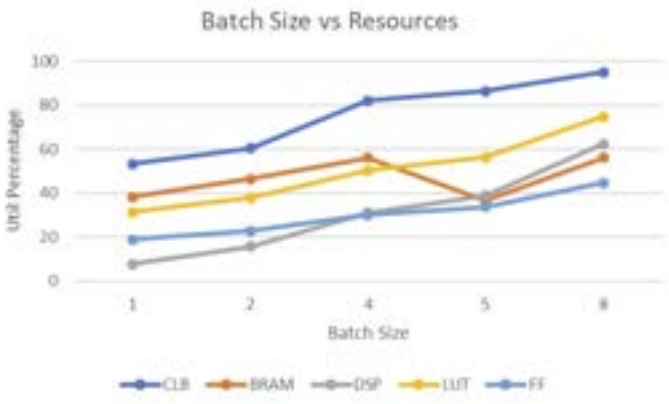

Figure 17: Resource Utilisation vs Batch Size

8 PROJECT MANAGEMENT

8.1 Schedule

Refer to our Gantt chart for our schedule. We have accounted for slack time and breaks as well.

8.2 Team Member Responsibilities

Initially, Abhinav and Rene worked on setting up the CPU benchmark and Yuhe worked on setting up the Ultra96 FPGA. Abhinav and Rene would then focus on algorithm analysis and learning about Vitis HLS optimizations while Yuhe set up our Vitis HLS workspaces. Towards the latter half of our project, we all focused on achieving our speedup goals. Nearing the end, Abhinav began to spend time deploying a web server that can visualize our simulation results (both live and asynchronously) and Rene helped with setting up the board's internet connectivity for the same.

8.3 Bill of Materials and Budget

Since Yuhe is taking 18-643 at the time of this project, we were able to borrow her 2 FPGAs (her board and her TA's) at no additional cost. We bought a MiniDP cable for around $9 on Amazon and also used a paid AWS EC2 t2.medium instance which cost us $0.8 (We ran this server for 10 hours for testing purposes and our final demo). We did not use most our allocated budget due to the software and digital hardware oriented nature of our project.

Bill of Materials		
Item	Quantity	Cost ($)
Xilinx Ultra96v2	2	0
MiniDP to HDMI Cable	1	8.54
AWS EC2 Instance	10 hours	0.8

8.4 Risk Mitigation Plans

One challenge we can face is running out of the programmable fabric's hardware resources. There are two types of resources we can run out of: computation resources and memory resources. The high-level synthesis compiler uses DSPs for calculation by default, but the Ultra96-V2 FPGA only has 360 DSPs, which limits the amount of computation we can do in parallel. In case we need more computation units, we can intentionally tell the compiler to use LUTs to perform some calculations, providing more computing capabilities. BRAM resources can also run out when we parallelize data access to a degree that uses up all the BRAM ports. To solve this issue, we need to rearrange our data accessing pattern, so that we partition data to fit BRAM block shape for efficient BRAM usage. If both strategies do not work, we need to parallelize less and improve speedup in other ways, such as increasing the frequency of our computation.

9 SUMMARY

To conclude, our project aims to accelerate N-body simulations on an FPGA. N-body simulations are computationally intensive simulations run by molecular and astronomical physicists and current implementations of this are slow and/or inefficient. We aim to leverage the FPGA's versatility and power efficiency to contribute to making these simulations more accessible to scientists on a budget while being sustainable for the environment. Our FPGA of choice is the Xilinx Ultra96v2, and hope to both accelerate and produce a visual representation of this simulation. Our plan for carrying out this optimization involves taking advantage of Vitis Pragmas like pipelining, and FPGA memory features like BRAM. While we do foresee complications with working with Vitis and some of the hardware restrictions of the FPGA (LUT count etc.), we are confident that we can come up with an optimal design to meet our design goals. [17]

Glossary of Acronyms

- ASIC - Application-specific Integrated Circuit
- BRAM - Block Random Access Memory
- CPU - Central Processing Unit
- DRAM - Dynamic Random Access Memory
- FPGA - Field Programmable Gate Array
- GPU - Graphics Processing Unit
- HDL - Hardware Description Language
- HLS - High-Level Synthesis
- LUT - Lookup Table
- DSP - Digital Signal Processor
- LUTRAM - Lookup Table Random Access Memory
- SRAM - Static Random Access Memory
- TDP - Thermal Design Power

References

[1] Wikipedia. *N-body Simulation*. Accessed: 15 December 2023. 2023. URL: https://en.wikipedia.org/wiki/N-body_simulation#:~:text=In%20physical%20cosmology%2C%20N%2Dbody,dynamical%20evolution%20of%20star%20clusters..

[2] T. BoekholtS.PortegiesZwart. *OntheReliabilityofN-bodySimulations*. Accessed: 15 December 2023. 2014. URL: https://arxiv.org/pdf/1411.6671.pdf.

[3] Avnet. *Ultra96-V2 Single Board Computer Hardware User's Guide*. Accessed: 15 October 2023. 2021. URL: https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf?MOD=AJPERES.

[4] Klaus Hinum. *N-body Simulation*. Accessed: 15 December 2023. 2023. URL: https://www.notebookcheck.net/Intel-Core-i7-9700-Processor-Benchmarks-and-Specs.477192.0.html#:~:text=The%20Intel%20Core%20i7%2D9700,clocked%20at%203%20%2D%204.7%20GHz..

[5] Takanori Sasaki Natsuki Hosono. *ParticleNumberDependenceof theN-bodySimulations*. Accessed: 15 December 2023. 2018. URL: https://iopscience.iop.org/article/10.3847/1538-4357/aab369/pdf.

[6] Nolan Foster. *What is the TDP of CPUs and GPUs*. Accessed: 15 December 2023. 2022. URL: https://www.acecloudhosting.com/blog/what-is-tdp-for-cpus-and-gpus/#:~:text=Thermal%20Design%20Power%20(TDP)%20for,power%20and%20generating%20more%20heat..

[7] Emanuele Del Sozzo et al. "A Scalable FPGA Design for Cloud N-Body Simulation". In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2018, pp. 1–8. DOI: 10.1109/ASAP.2018.8445106.

[8] TOM VENTIMIGLIA KEVIN WAYNE. *The Barnes-Hut Algorithm*. [Online; accessed 2nd October 2023]. URL: http://arborjs.org/docs/barnes-hut.

[9] Raphael K. *Barne's Hut Algorithm*. Accessed: 15 December 2023. 2013. URL: http://15418.courses.cs.cmu.edu/spring2013/article/18.

[10] Murali Ravi et al. "FPGA as a Hardware Accelerator for Computation Intensive Maximum Likelihood Expectation Maximization Medical Image Reconstruction Algorithm". In: *IEEE Access* PP (Aug. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2932647.

[11] wordchao. *HLS Optimization: Throughput*. cnblogs. 2019. URL: https://www.cnblogs.com/wordchao/p/10943227.html.

[12] AMD Adaptive Computing Documentation Portal. *Versal ACAP Configurable Logic Block Architecture Manual (AM005)*. Accessed: 14 October 2023. 2023. URL: https://docs.xilinx.com/r/en-US/am005-versal-clb/Look-Up-Table.

[13] Kinsta. *Flask vs Django: Choosing the Best Python Framework*. https://kinsta.com/blog/flask-vs-django/. Accessed: 2023-12-15. 2023.

[14] Simplilearn. *AWS vs Azure: Which Cloud Platform Should You Choose?* https://www.simplilearn.com/tutorials/cloud-computing-tutorial/aws-vs-azure. Accessed: 2023-12-15. 2023.

[15] Amazon Web Services. *Amazon EC2 Pricing - On-Demand*. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2023-12-15. 2023.

[16] Mike Bailey. *Parallel Programming: Speedups and Amdahl's law*. Accessed: 15 December 2023. 2021. URL: https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.1pp.pdf.

[17] Intel® FPGA SDK for OpenCL™ Standard Edition: Best Practices Guide. *Floating-Point versus Fixed-Point Representations*. Accessed: 14 October 2023. n.d. URL: https://www.intel.com/content/www/us/en/docs/programmable/683176/18-1/floating-point-versus-fixed-point-representations.html.

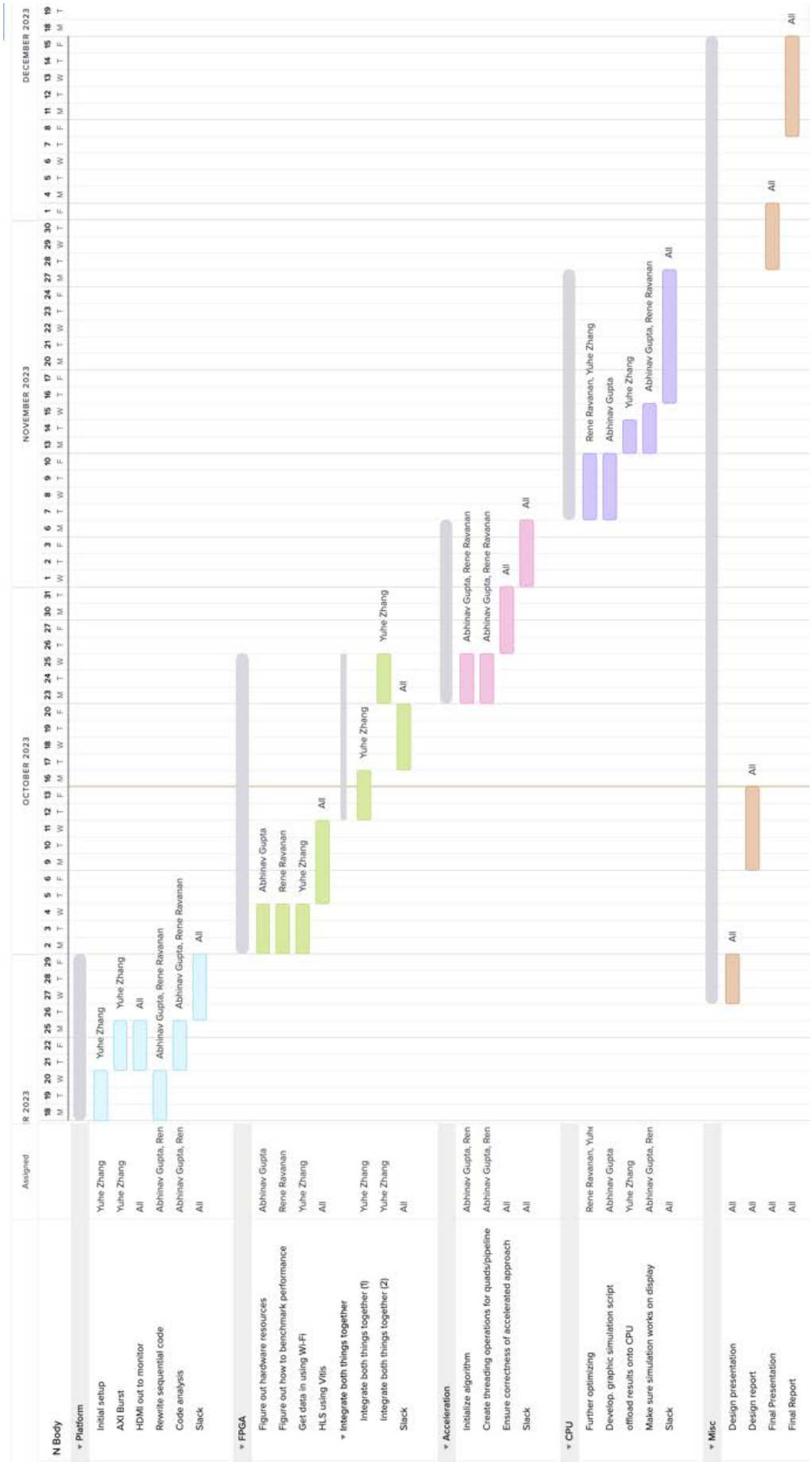

Figure 18: Gantt Chart