FPGA Accelerated Fluid Simulation

Authors: Jeremy Dropkin, Alice Lai, Ziyi ZUo
Affiliation: Electrical and Computer Engineering, Carnegie Mellon University

For this project, we design a system capable of running the fluid simulation kernel developed for the Scotty3D graphics library on the Xilinx Ultra96 platform. The original implementation is significantly bottlenecked by the limited concurrency of multithreading and data transfer, so we believed that it would be a suitable target for a form of hardware acceleration that could exploit the inherent parallelism offered by the program. We epxloit the customizable nature of FGPAs to greatly improve the speed at which frames are rendered, and managed to achieve a 100x speedup in the render kernel.

***Index Terms*—Acceleration, Fluid Simulation, FPGA, High Level Synthesis, Ultra96, Vitis, Vivado, Xilinx**

1 INTRODUCTION

Fluid simulation is one of the most popular computer graphics applications for FPGA acceleration. Fluid simulation is uniquely suited for FPGA acceleration due to its high workload and high irregularity in computation. As an example, a common task in fluid simulation is to handle collisions between neighboring particles and the scene geometry, which very easily is an $O(n^2)$ task. However, a recent paper published in 2013 by Macklin et al. was able to showcase an alternative approach to fluid simulation, one free from much of the heavy computation that plagues conventional fluid simulation algorithms, position based fluids. Essentially, this new approach enforces a set of positional density constraints to simultaneously allow for incompressibility (the computationally expensive part of conventional techniques) and for large time steps and stability.

Our project was based on Scotty3D, a custom 3D graphics software package developed for 15-462 Computer Graphics course at CMU, that features position based fluid simulation. Scotty3D features basic multi-threading as it is designed to be accessible to students who may not have access to a GPU, either on their laptop on their PC machine. Unfortunately, this multi-threaded, CPU-only implementation of Scotty3D does not yield impressive performance, taking up to 3 seconds to process a single frame of the fluid simulation. This leads to choppy and visually unsatisfying simulation results. As such, we propose our project as a solution for accelerating and improving the user experience of the Scotty3D fluid simulation engine.

One competing solution in our acceleration space is the use of a GPU to accelerate this fluid simulation algorithm. On one hand, while the GPU benefits from the high level of thread-level parallelism, it suffers in the fact that it necessitates highly vectorized code, which may suffer some overhead from needing to program the different branching conditions. Compared to the GPU, an FPGA is much more flexible in terms of different branching behaviors, yet still retains the capability of high thread-level parallelism. Another competing technology is a custom ASIC implementation. While such a device would be expected to greatly outperform the FPGA, its use is simply out of the picture during a capstone semester.

2 USE-CASE REQUIREMENTS

Our main use-case requirements are about the metrics of our fluid simulation. We want to ensure that we speed up the simulation, but additionally that we maintain accuracy and are able to simulate a reasonable size of particles. These goals seem like they may be in contention as generally the larger a problem is, the longer it will take to simulate, additionally, if something is faster, it is likely to lose accuracy. So it is important in our project that we make requirements and optimizations that allow us to keep all of our requirements achievable and do not interfere too greatly with each other.

Our first use-case requirement is that we want to achieve a 10x speedup with our accelerated version when compared to a base implementation on a CPU. The motivation for this requirement is from initial measurements that we took on an i7 CPU and found that it takes roughly 3 seconds to produce each iteration of the simulation for a size of 512 particles; this measurement is the same as 0.3 fps. We felt that achieving a 10x speedup is realistic, as the fluid simulation problem is well suited for FPGA acceleration. So being able to run the simulator at a rate of producing an update iteration every 0.3s, i.e. 3fps would greatly increase the utility of the rendering engine.

Next, we are targeting support for simulating 512 particles at once in our rendering engine. We arrived at this number of particles after considering several factors, and decided that this was the most practical number to target, and is a useful number of particles for a user. 512 particles is the default and typical simulation size for the Scotty3D simulator, so this would allow us to easily create benchmarks to measure against as many of the scenes present in Scotty3D are this size. This number of particles also reflects well to the specs of the hardware we are using.

Finally, we want to ensure that the accuracy of our simulation is high enough to produce realistic output. While we want to produce output quickly, that output would be useless if it was not an accurate simulation. We will evaluate accuracy using the Chamfer Distance (CD) metric, which finds the nearest point in the other point set and sums the squared distances. We will be comparing the frames that we produce from the accelerated simulator to the accurate, but slow, CPU simulator. We are targeting a maximum Chamfer distance of 2.56 for any given frame.

3 ARCHITECTURE AND PRINCIPLE OF OPERATION

We implemented our solution with the standard accelerator architecture where a host computer dispatches acceleration tasks to a dedicated piece of acceleration hardware. For this project, we are using the Xilinx UltraScale+ Ultra96v2 development board, which features both an ARM core for running C++ code and a flexible FPGA fabric for accelerated kernels. We had initially planned to run the entire Scotty3D environment on the FPGA, but after initial testing we pivoted to having the FPGA being a dedicated piece of acceleration for just the fluid simulation algorithm.

The high-level approach of our project is to isolate the Scotty3D fluid simulation kernel to run on the board's FPGA fabric, and have a host computer manage launching the simulation kernels, which includes handling the data transfer to and from the fabric and a network-related cost to scp the file between the FPGA and the host computer. The simulations will produce output files which can be used either for post-processing or to view an animation of the simulated output. We implemented a modified version of Scotty3D that is capable of displaying these files.

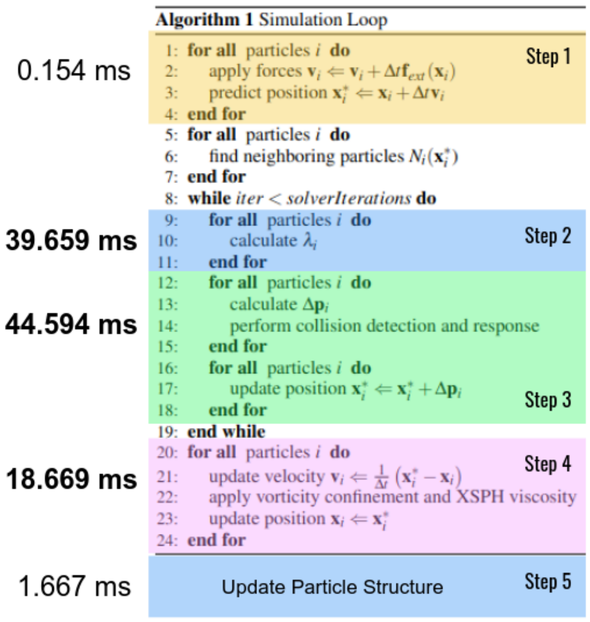
As illustrated in Figure 1, the CPU handles the initial fluid simulation request and sends a signal to the FPGA fabric controller. This then tells the board to execute the main fluid simulation kernel. The CPU will also initiate a burst data transfer from main memory, which will sends the initial scene data and all of the fluid particles to the Block RAM (BRAM) modules of the FPGA. We did all of our simulation updates with the BRAM modules, which get updated in-place by the main fluid simulation kernel.

Once the algorithm has completed a sufficient amount of density iterations to perform a single step in the main fluid simulation loop, the updated particles are returned to the CPU. The CPU then finally takes the updated particles and saves them to the output file. This loop runs for an indefinite number of iterations while producing the simulated fluid output.

The main fluid simulation algorithm is split up into five discrete steps, and this is where the bulk of the acceleration will take place. These steps generally consist of addition, convolution, multiplication, dot products, etc. to all of the particles being simulated, as well as data transfer between different data structures storing different physical information. These steps will all be running on the FPGA fabric on the board, and will be dispatched to by the CPU.

By taking advantage of the FPGA in conjunction with the ease of writing code for a CPU, we are able to focus our efforts on speeding up only what needs to be, and still have the ability to write non-highly optimized code for the controller on the CPU. This setup allows us to maximize the time we spend on actual acceleration, instead of having to focus on getting non-critical parts of the algorithm to run on the FPGA fabric.

4 DESIGN REQUIREMENTS



Algorithm 1 Simulation Loop

```
1: for all particles $i$ do
2:     apply forces $\mathbf{v}_i \Leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_{ext}(\mathbf{x}_i)$
3:     predict position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
4: end for
5: for all particles $i$ do
6:     find neighboring particles $N_i(\mathbf{x}_i^*)$
7: end for
8: while $iter < solverIterations$ do
9:     for all particles $i$ do
10:        calculate $\lambda_i$
11:    end for
12:    for all particles $i$ do
13:        calculate $\Delta\mathbf{p}_i$
14:        perform collision detection and response
15:    end for
16:    for all particles $i$ do
17:        update position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i^* + \Delta\mathbf{p}_i$
18:    end for
19: end while
20: for all particles $i$ do
21:     update velocity $\mathbf{v}_i \Leftarrow \frac{1}{\Delta t}(\mathbf{x}_i^* - \mathbf{x}_i)$
22:     apply vorticity confinement and XSPH viscosity
23:     update position $\mathbf{x}_i \Leftarrow \mathbf{x}_i^*$
24: end for
```

Figure 2: Timing for CPU (i7) Base Implementation

4.1 Steps 2 and 3: Scale Factors & Position Correction

Step 2 is responsible for calculating the scale factors for the Newton's Method optimization loop, and Step 3 is responsible for calculating the position correction for each particle, as well as collision detection and adjusting the position corrections accordingly. Our first design requirement is that we want to achieve at least a 10x speedup of Steps 2 and 3 combined, and get the time to less than **5.583 ms**.

4.2 Step 4: Vorticity & Viscosity

Vorticity is the local angular rate of rotation, or the amount of "circulation" in a fluid. Viscosity is the measure of resistance to flow. To keep the fluid from exploding, the algorithm subtracts the vorticity term and adds the viscosity term to each particle's next velocity. Our second design requirement is that we want to achieve a 4x speed up, or get the time to less than **4.667 ms** for calculating XSPH viscosity and vorticity confinement corrections.

4.3 Steps 1 and 5: Position Management

Steps 1 and 5 are responsible for the initial position and particle data structure updates respectively. The compute

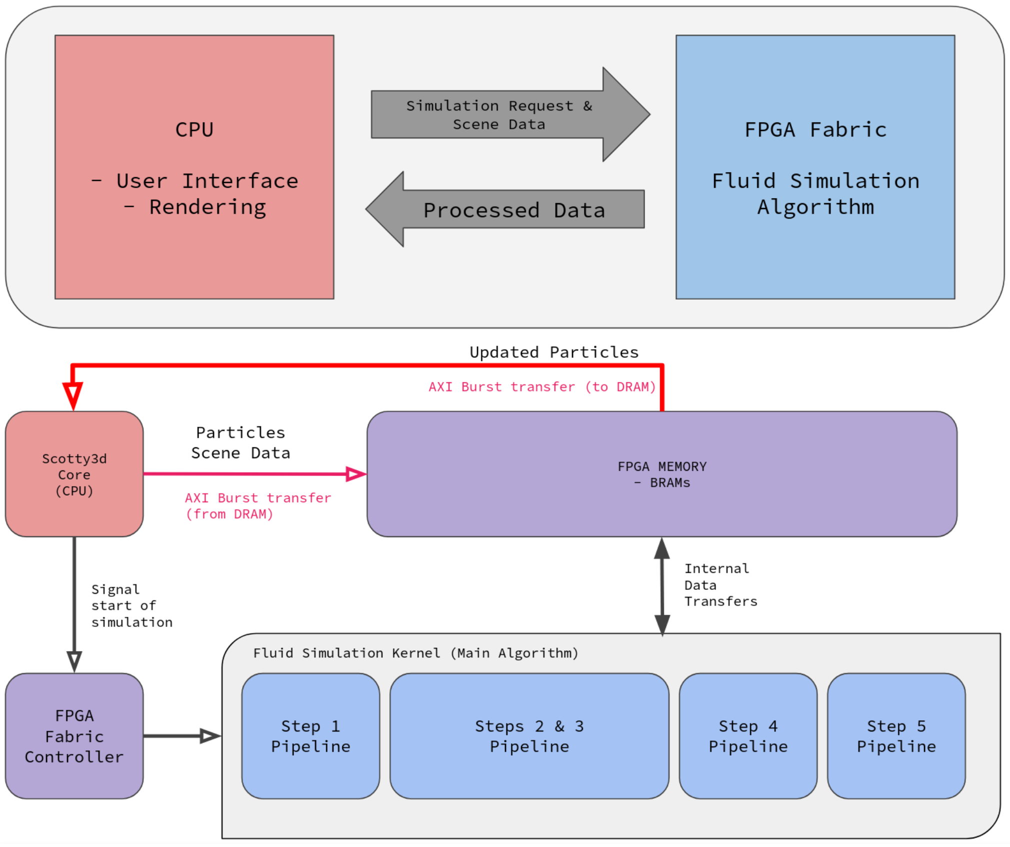


Figure 1: (top) High Level Block Diagram — (bottom) Detailed System Diagram

time is almost negligible compared to the previous target optimization, but nonetheless our final design requirement is that we want to achieve at least a 2x speed up for these two steps, or less than **0.910 ms**. This target essentially is to ensure that the data transfer speeds on the FPGA with BRAM are significantly faster than on CPU with DRAM and cache memory.

4.4 Synthesizability

Something that was hugely important to our project but somewhat initially overlooked in our design report was the aspect of actually writing code that fits on the FPGA. Since CPU code tends to assume that there is a nearly unbounded amount of memory, and doesn't have to worry about meeting clock slack, when porting the code we had to take all of this into account. So a potentially obvious but large requirement is to actually write code that is able to synthesize onto the FPGA.

5 DESIGN TRADE STUDIES

As with any acceleration endeavor, approaching this project required us to make many careful considerations as to the different directions by which we should approach the project. For our project design, we considered the target platform, the consequent hardware acceleration approach for synthesizing to an FPGA, and finally the fluid similarity metric for evaluating accuracy.

5.1 Hardware Platform

Our first major consideration was the hardware platform that we would build the project on. Different computational platforms each have their own pros and cons, and to efficiently accelerate this project, we would need to assess the workload of Scotty3D fluid simulation kernel in order to determine a suitable platform. Some of the different platforms we considered but ultimately rejected include:

5.1.1 *Purely CPU - Multithreaded*

The most basic approach to acceleration is to just use a multithreaded CPU implementation. However, we believe that the most natural avenue for parallelism is to apply it on the axis of particle number. Since we are working on the order of hundreds or even thousands of particles, the scope of multithreading is unable to fully capture the parallelism inherent in the program. Furthermore, we could also stand to pay significant costs through inter-thread communications and thread startup and synchronization costs.

5.1.2 *GPU*

So, if high concurrency is what we want, then surely it would make sense to put the algorithm on a GPU? After all, GPUs are already a common target for graphics acceleration tasks. The issue with the GPU implementation is that the fluid simulation kernel is highly irregular. There are many data-specific loops that operate on the nearest neighbors of a particle. Due to how divergent the control flow is for different threads, we could expect to pay significant overhead costs in realizing the algorithm on a GPU. Furthermore, since the fluid simulation algorithm is highly dependent on data movement, the memory architecture of traditional GPUs leaves a bit more to be desired.

5.1.3 *ASIC*

Although an ASIC would be very efficient for any application, unfortunately we do not have the proper amount of or time in order to send our design off to a chip fab. In order for an ASIC to be worth it we would need to be producing our design at scale, and for a prototype system this is not a good choice.

5.1.4 Why FPGA?

Ultimately, we decided to use an FPGA as the target platform. FPGAs are applicable in our use for many reasons - the customizability of the hardware allows us to exploit massive parallelism potential in the fluid simulation algorithm, and they are relatively cheap compared to ASICs. There is also a lot of well-documented tooling available for accelerating code on FPGAs, so using an FPGA makes a lot of sense given our application. FPGAs are also well suited towards irregular computation since the hardware does not have to be uniform, since we can just instantiate extra hardware to account for the different cases. Our algorithm is non-uniform as particles can be clustered in random configurations, so using an FPGA helps us deal with this non-uniformity. Another beneficial aspect of FPGAs is that they have great flexibility in their memory architecture. By making use of the BRAM banks on the FPGA, we are better able to exploit memory reuse and reduce the number of high-latency trips to and from DRAM.

5.2 Hardware Acceleration Approach

5.2.1 *HLS vs. Traditional HDL*

Once we decided to use the FPGA, we needed to decide whether we would stick with using a traditional hardware design language like SystemVerilog to write our kernel or whether we would use high level synthesis. Ultimately, our decision to use HLS was motivated first by the fact that Scotty3D is already written in C++, so porting it to something that works with HLS is more approachable than rewriting it entirely in SystemVerilog. This means we have more time to focus on actually accelerating the algorithm, rather than reimplementing existing code in a different language. Additionally, many HLS tools make parallelization more accessible than if a traditional HDL is used, as the compiler is able to smartly infer ways to optimize code from a higher level that would not be possible from SystemVerilog.

5.2.2 *Fixed-Point vs Floating-Point Numbers*

One major decision for our system was choosing whether to use fixed or floating point numbers to represent positions of particles. Although using floating point numbers allows for arbitrary amounts of precision, they are much more costly to do computation on. We also examined how the fluid simulation application uses, floats, and only roughly 3 decimal digits of precision are needed to maintain accurate computation, so 16.16 fixed-point numbers are more than precise enough. As we can see in Table 1, the latency of a fixed-point add is about 4.5 times faster than a floating point add.

Ultimately, we ended up using 12.6 fixed-point numbers, as the precision of 16.16 was actually greater than what we needed, so we were able to shrink the size of the numbers and take more advantage of the BRAMs to further accelerate our kernel.

5.3 Fluid Similarity Metrics

Because the fluid is composed of discrete particles, the fluid can actually be considered as a point cloud. There exists many distance metrics for point clouds in computer graphics and robotics literature. We provide a high-level overview of some popular distance metrics we considered as well as their strengths and weaknesses as followed.

5.3.1 Earth Mover's Distance (EMD) / Wasserstein Distance

The Earth Mover's Distance (EMD) is another metric for measuring the distance between two distributions of points. Colloquially, this distance metric views each distribution as a unit of soil, and measures the minimum cost of transforming one pile into the other. The cost is calculated by multiplying the "amount of earth" with the "average distance moved". More formally, for two probability measures μ and v, the distance d_{EMD} is defined as:

$$d_{EMD}(S_1, S_2) = \min_{\phi : S_1 \to S_2} \sum_{x \in S_1} \|x - \phi(x)\|_2$$
$$\text{where } \phi : S_1 \to S_2 \text{ is a bijection.} \quad (1)$$

The EMD is the Wasserstein Distance using one-dimensional distributions, i.e. $p = 1$. Applied to our scenario in $\mathbb{R}^3$, this would mean that we would have to pick one axis to evaluate the point distributions along. While it is tempting to use the Wasserstein Distance with $p = 3$ given the $\mathbb{R}^3$ nature of the problem, computer graphics literature

Table 1: Fixed-point vs Floating-point Metrics

	Addition Latency (ns)	Multiplication Latency (ns)
Floating	7.717	10.432
Fixed	1.692	4.369

tells us that the computational cost associated with it is too high. For reference, see below for the formal definition of Wasserstein Distance:

$$W_p(\mu,\nu) := \left(\inf_{\gamma\in\Gamma(\mu,\nu)} \int_{M\times M} d(x,y)^p \, \mathrm{d}\gamma(x,y) \right)^{1/p} \quad (2)$$

5.3.2 Chamfer Distance

The Chamfer distance metric is a rather straightforward one. For every point in Point Cloud A, it finds the nearest point in Point Cloud B and sums the squared distances. It then goes the other way and sums the squared distance between the points in Point Cloud B and their nearest Point Cloud A points. Mathematically, the Chamfer Distance d_{CD} is defined as:

$$d_{CD}(S_1,S_2) = \sum_{x\in S_1} \min_{y\in S_2} \|x-y\|_2^2 + \sum_{y\in S_2} \min_{x\in S_1} \|x-y\|_2^2 \quad (3)$$

5.3.3 EMD vs. Chamfer Distance

The EMD and Chamfer Distance metrics are the two most popular distance metrics using in computer graphics algorithms at the moment. Though EMD allows for more accurate learning-based 3D reconstructions, the Chamfer Distance is generally favored over the EMD however due to light computational cost.

Since the fluid simulation is comprised of multiple frames, i.e. multiple pairs of point distributions, we would need to evaluate the distances for each frame (which for the baseline fluid simulation is about 10), so we decided that the computational cost was more important than increase in accuracy and chose the **Chamfer Distance** metric.

6 SYSTEM IMPLEMENTATION

6.1 Pipelining

One key benefit of using an FPGA, and a way that we can increase the throughput of our system, is by using pipelining. Pipelining consists of splitting up steps of a process between registers and increases hardware utilization by concurrently having different operations use the various regions of hardware at once. By making sure that all the functional units are busy, we maximize the work done by the hardware, thereby increasing throughput of our kernel.

When we render each frame, each point needs to undergo several mathematical operations, and these must be done sequentially to ensure the algorithm is correct. But, it is possible to have some points using the hardware for one mathematical operation, while other points are using undergoing another operation. See Fig. 3 for an example of pipelining - each set of boxes represents an entire task, so without pipelining an entire task must finish entirely before the next one starts, but with pipelining, independent stages of a task can execute concurrently thus improving throughput. Specifically in our system, steps 2 and 3 are easy to pipeline, as they work on the same set of particles, and are sequential mathematical operations.

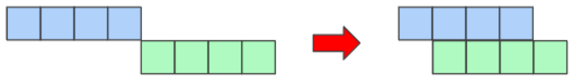


Figure 3: Pipelining

Nevertheless, when it comes to pipelining, we will both be limited by the algorithm itself. Specifically, there are some steps that have inherent data dependencies, where new tasks cannot be initiated due to relying on data from previous tasks. As such, we cannot pipeline between certain steps, as certain steps will require all the data for the previous steps to be calculated for **all** of the particles.

6.2 Loop Unrolling

When running a loop on a single-core, non out of order CPU, each iteration of the loop must run sequentially, even if there are no dependencies between iterations of the loop as the CPU can only run one instruction at a time. When running in hardware, the equivalent of a loop on a CPU can run easily in parallel; it is as simple as creating more hardware to do multiple computations of the loop body in parallel. See Fig. 4 for an illustration of unrolling - in this figure time is passing from left to right, so initially all the iterations run sequentially, but in the final stage all of the iterations run at once. In the fluid simulation algorithm, there are many places where it loops over all particles, and does some mathematical transform on them. We are able to exploit massive parallel gains here, as we are able to do these computations at the same time for many points, thereby reducing the time for each loop to complete by a large factor.

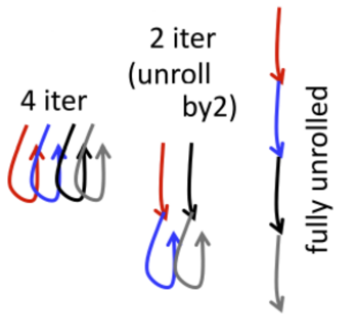


Figure 4: Loop Unrolling

6.3 Fixed-Point Numbers

The fluid simulation algorithm we are accelerating uses a lot of fractional numbers as positions that particles are at need to be tracked accurately. The distance between particles is important to have as precise as possible to end up with the best results. Using floating point numbers was our initial thought for how to represent these numbers, as in nearly any CPU-based implementation floats would be the default. But, when speed is priority floats are slow. Floating point numbers use some of their bits to store the position where the fractional portion of the number starts (The exponent portion of the top diagram in in Fig. 5). This means when doing operations on two floats, their floating points must be aligned, and this is expensive. An alternative is to use fixed-point numbers, these numbers do not store where the fractional portion starts, and instead the have a fixed number of bits for the integral part and the fractional part. This can lead to numbers being less accurate, but for our case we do not ever need more than 16 bits of fraction or integer. Fixed-point numbers are much faster as they can just be represented as integers, so doing many common operations are nearly as fast as, or exactly as fast as doing computations on a normal integer.

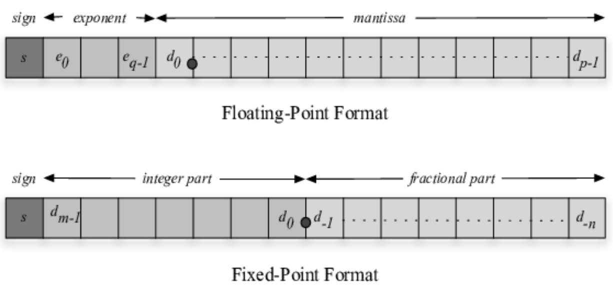


Figure 5: Floating-Point vs Fixed-Point Numbers

6.4 FPGA Memories

6.4.1 DRAM

DRAM will be the main memory that our system is working with. All of the non-accelerated parts of the algorithm will be using their default DRAM configurations. That is, we will simply of the operating system and the C++ compiler handle all of the memory interfacing here. However, in terms of the fluid simulation kernel itself, we want to make sure that we make as few trips to main memory as possible. This is due to the fact that, compared to the following two memory implementations, the DRAM latency is hundreds to thousands of cycles longer. Since we do not want to incur those penalties, we want to isolate our interactions with DRAM to just getting the data at the start of the computation and sending data back at the end. *We should note that aside from the control signals between the FPGA and the CPU, DRAM will be the main point of communication and data transfer for our acceleration task.*

6.4.2 LUTRAM

The next memory of importance is the LUTRAM, which is implemented using the multitudinous Lookup Tables (LUTS) scattered around in all of the combinational logic blocks in the FPGA. Compared to DRAM, LUTRAMs typically take just a single clock cycle of latency. However, they are typically better suited for storing small collections of data, as these memories are not dense in terms of bits/area. As such, they will better be suited for storing constants such as gravity or any variable figures that we might use in our calculation.

6.4.3 Block RAM

Block RAMs (BRAMs) are the major ace up our sleeve in terms of acceleration on FPGAs. Like LUTRAMS, BRAMs typically have just a single cycle of latency. They are typically implemented as a collection of various SRAM banks with some additional functionality (e.x. true double-porting). Not only are BRAMs incredibly fast for their size, they are also quite data-dense, and widely reconfigurable in terms of their addressing modes and their access patterns. For our project, we will be using the BRAMs to store the fluid simulation particles. This is a natural fit, as we will be dealing with hundreds of distinct particles, and editing their attributes throughout the runtime.

6.5 C++ Dispatcher

Since we are only working on optimizing the heavy computational part of the fluid simulation, the main dispatcher of the algorithm will run in C++ on the CPU on the board. This allows us to dynamically dispatch commands through an AXI control interface to the compute kernels, so when inputs are ready they can be sent to the fabric, and are then sent to the next stage by the CPU.

As mentioned earlier, we did not have the time to fully port the entire Scotty3D stack to the FPGA, so the C++ dispatcher does still exist, but in a more limited form than we intended. Now we require using SCP for getting the outputs from the board, and user experience is less friendly than we had hoped.

6.6 Nearest Neighbor Lookup

For the fluid simulation kernel, one task that we will be performing a lot is the nearest neighbor lookup. In the original implementation, this was done with a hash-map that used a quantized three-dimensional position to lookup an array of neighboring particles. This provides a O(1) lookup with an O(n) update. In order to keep this performance in hardware, we needed to be a bit more creative.

We implemented the neighbor lookup by using a bounded 3D voxel space of particles. The entire simulation space is quantized into smaller cubes, and we can quickly find which cube a particle is in based on its ID and position. Then, in order to find a particle's neighbors, the algorithm simply just looks at all of the neighboring voxels to the voxel the particle is in, and any particles in those voxels are counted as neighbors. Initially we set out to use a "hardware hash map" to implement the neighbor lookups, but we quickly realized that the 3D voxel space was both simpler and more efficient to implement.

It's worth noting that while our approach could sacrifice some of the accuracy or scope of the simulation, we believe it is worth doing so that we can avoid the incredible expensive trips to DRAM. Another limitation of our approach is that since the simulation space is bounded, if particles end up moving too far they are required to be dropped from the scene.

7 TEST & VALIDATION

7.1 Speedup Results

Changes	Latency (Cycles)	Cycle Time (ns)	**Kernel Runtime (ms)***	**Timed (ms)**
Baseline CPU	-	-	**-**	**15000**
Baseline	4,552,408,073	10	**4,552.41**	**602.66**
Unrolling	4,550,196,233	10	**4,550.20**	**602.61**
Pipelining	264,654,053	10	**264.65**	**273.73**
Both	200,889,533	10	**200.89**	**250.03**

Figure 6: Performance increases from optimizations (5 frames for CPU; 10 frames for FPGA)
*Reported by Vitis HLS

In the table above, we see the timing results of rendering 5 frames on the CPU and 10 frames on the FPGA. Observing the table in Figure 6, as we apply the optimizations described in the System Architecture section, we see a decrease in the worst case runtime of our kernel. From our results, we determined that enabling pipelining was the most effective optimization we made. Still, we can see that even without any micro-optimizations, we can still achieve a 50x speedup over the original CPU implementation. The spectre of Amdahl's Law (which discusses the performance gain relative to what can be optimized) still looms over us, as our optimizations were eventually bottlenecked by data transfer bandwidth limitations.

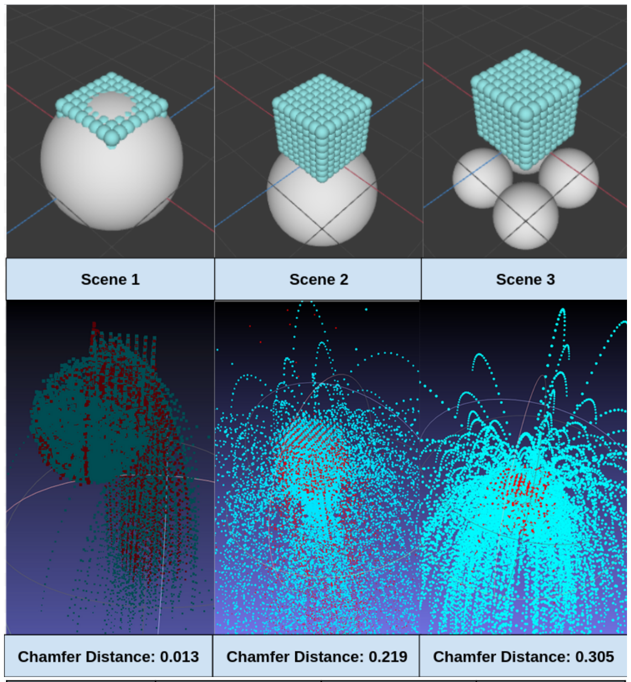
Finally, observing the full runtime (which includes the the time from actually running the kernel on a set of 512 points as well as exporting the results from the FPGA to the Host CPU, we can see that With all optimizations applied, our speedup from the original CPU baseline is a 100x speedup to render a single frame, which is much larger than our initial goal of 10x. In order to fully reach this speedup, we utilized nearly all of the resources available on the FPGA. In fact we actually hit the largest possible extent in speedup, as we had maxed out our LUT utilization at 99%.

7.2 Accuracy Results

Below are accuracy results for 3 different scenes. The blue particles are the reference fluid particles, and the red particles are the FPGA output fluid particles.

The first scene is without much motion since the fluid and the sphere are already colliding, so the 3D fluid simulation traces are very similar and the Chamfer distance is extremely small. In the 2nd and 3rd scenes, you can see a lot more blue than red, which makes sense since we drop the particles in our FPGA implementation outside of a fixed voxel space. The Chamfer distances are still quite reasonable.

Visually, the FPGA-generated fluid simulations in all three scenes looked reasonable and fluid-like. All of the scenes met the Chamfer distance requirement (less than 2.56). Therefore we have fully met our design requirements for accuracy.



	Chamfer Distance (m)	Particle Diameter (m)	Ratio (PD/CD)
Ours (Scene 1)	0.013	0.125	9.615
Ours (Scene 2)	0.219	0.125	0.570
Ours (Scene 3)	0.305	0.125	0.41
Reference	0.03	0.05	0.6

Figure 7: (top to bottom): Scene, Fluid Simulation Output, Table of Particle Diameter to Chamfer Distance Ratios

7.3 Limitations

7.3.1 Scaling

One limitation of our project in terms of how we investigated the speedup is that we focused our speedup efforts on a fixed number of particles. In common literature, we focused on investigating the effects of *strong scaling*, which refers to the effects of accelerating a tasked **without** also scaling up the amount of input data. Typically, for this context, we analyze the results from the viewpoint of **Amdahl'S Law**, which we saw above.

However, some potential future work for this project could be to consider the effects of speedup when we also consider increasing the number of particles. In the current iteration of the project, we did not pursue this task due to time limitations, but we are confident enough in the results of the algorithm to provide analysis as to what should happen (and why we would expect our FPGA implementation to benefit more from weak scaling than the CPU implementation).

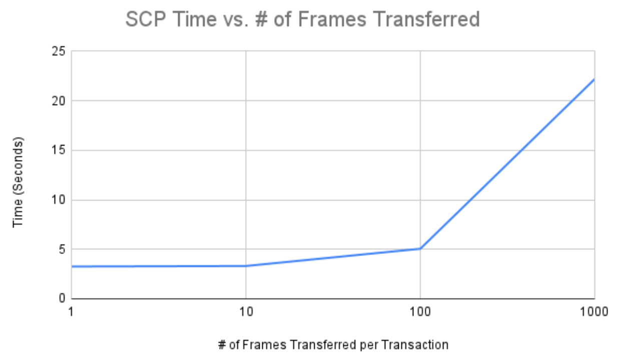
For the compute side of scaling, we can expect that the number of computations performed in both the CPU and the FPGA version of the algorithm will scale equally. Although it should be noted that the FPGA would continue to have the benefit of concurrency over the CPU, ensuring it some degree of linear speedup over the CPU.

On the memory side of scaling, as stated above, previous analysis of the CPU version of the kernel indicated that it was heavily memory bound, where a large majority of the time spent in the kernel was spent updating the nearest neighbor hashmaps. As such, as we increase the simulation size, the sizes of the arrays that the hashmap point to must also increase as well, which could incur some heavy costs, given how C++ implements arrays and hashmaps. Any any sense, growing the sizes of these data structures while abiding by their structural laws will incur a superlinear performance degradation. On the other hand, for the FPGA side, whenever we build a kernel for a maximum neighbor size, we have already committed to supporting more particles and more particle storage. So, there is little additional overhead to supporting more particles aside for a 1) linear increase in the writeback buffer for the kernel and 2) a $\frac{Scale\ Factor}{\#\ of\ Voxels}$ increase in the length of the nieghbor arrays for the various entries in the voxel space.

In the end, while the compute scaling will likely be the same for the CPU and the FPGA when it comes to weak scaling, the CPU will be more significantly bottlenecked by the scaling of the working set due to the structure of its memory hierarchy. As such, we expect the FPGA to still outperform the CPU when it comes to scaling up the number of particles.

7.3.2 Data Transfer

For our project, we needed a method to transfer the simulation results from the FPGA board to the client machine. As the focus of our project was to accelerate the fluid simulation kernel, we just decided that we would stick with just using SCP to copy the fluid simulation text from the board. However, if we were to scale up this project, we would quickly find that SCP would not be able to keep up with the data transfer rates we would require. If we observe the following table, we may see the SCP times for transferring 1, 10, 100, and 1000 frames of data at a time



From the graph, we can see that there is little difference between 1 and 10 frames at a time; a small increase from 10 to 100 frames at a time, and a large increase from 100 to 1000 frames at a time. The reason 1 frame and 10 frames have similar times is likely because these are a small number of frames, so the runtime of both are likely just dominated by the overhead costs of setting up the SCP transaction in the first place. The difference between 10 and 100 just seems to be a fairly regular linear increase. However, the jump from 100 to 1000 frames is fairly significant. We believe that this is likely due to the fact that the jump in packet size between 100 and 1000 frames (1.2MB vs 12MB, respectively) likely crosses some boundary in the memory hierarchy. For instance, perhaps 1000 frames happens to be larger than the size of any of the cache boundaries of the core on the chip. As such, the 1000 frame transfer might have to incur many expensive trips to DRAM than the 100 frame transfer.

Either way, transferring 1000 frames of simulation at a time is a fairly significant task. Ideally, such a transaction would be bursted into many different batches, rather than a single batch. Nevertheless, this still demonstrates a potential downfall of our approach, and a potential boundary in which it would be more performant to just perform all of the computations on the CPU.

8 PROJECT MANAGEMENT

8.1 Schedule

See last page for our Gantt chart of responsibilities.

8.2 Team Member Responsibilities

Initially Alice and Ziyi were assigned to work on the acceleration of the core algorithm, and Jeremy would help with auxiliary features and issues and benchmarking. However, we realized that it would be better if Alice and Jeremy swapped, since we realized benchmarking relied on adding new features to Scotty3D, which Alice was better suited for given her prior experience with Scotty3D, and acceleration of the core algorithm was better suited for someone with good knowledge of hardware, which Jeremy had/

8.3 Bill of Materials and Budget

Because of the software and digital hardware oriented nature of our project, we already have all the materials necessary and will not be spending any money. The target FPGA platform is an Ultra96 v2 borrowed from 18-643.

8.4 Risk Management

As mentioned earlier in the report, we were originally going to support the entire Scotty3D stack on the FPGA (i.e. rendering, UI, etc.) but during our initial set-up phase we realized that this was too high of a workload for the Ultra86, so we had to change our system architecture to accommodate the FPGA being only a compute unit that could pipeline with a host computer to run the rest of the Scotty3D stack.

Getting the base kernel to work also took much longer than we expected, and we had to use up a lot of our slack in the project schedule and get rid of nice-to-have features (such as a request scheduler) that we wanted to include towards the end.

We were going to extend our implementation to be able to support arbitrary meshes as well, but we quickly realized that that necessitated compiling OpenGL on the Ultra96. That alone took a couple days just to try to get it installed on the Ultra96, but in the end we were not able to make much progress, and so we scaled back to only supporting sphere primitives.

9 ETHICAL ISSUES

Our project likely has two use cases, one is computer animation, and the other would be for scientific computation. There are different ethical dilemmas that can stem from both areas, with some being more significant than others. For scientific computation, if there is a calculation that relies on the results of a fluid simulation in order to make a judgement, for example how thick to make a dam, an inaccurate simulation could end up causing catastrophic consequences. Or even if it is something less drastic, being able to run more simulations in a given time frame would allow for more results to be produced, and therefore potentially make scientific contributions in a shorter time span.

For animation there are more monetary concerns than those of danger - if a fluid simulation is able to run faster, it is possible to produce a product in less time. So either a better quality final product can be made, or the quantity of output can be grater, and depending on the goals of who is using the fluid simulator, both of these possibilities have large monetary implications, and how money is allocated comes with a whole heap of ethical concerns of its own.

10 RELATED WORK

There exist a plethora of FPGA-accelerated fluid simulation papers, which adapt and optimize kernels that perform traditional fluid simulation functions such as advection, approximation of divergence and curl, etc. The difference between our project and the rest is that no others (to our knowledge) target position-based fluid simulations.

11 SUMMARY

Our system was able to meet our design requirements. We are able to achieve a 100x speedup, where our requirement was to have a 10x speedup. The system is limited in that it cannot perform fluid simulation outside of the bounding voxel space, but that unfortunately is tied to the resources available on the FPGA platform.

11.1 Future Work

Because we wanted to preserve fast neighbor lookups, we were forced to constrain the voxel space to a small 4 x 4 x 8 unit cube. In the future, we would like to explore using a larger FPGA platform that could allow for a larger voxel space, and explore alternative approaches and data structures for keeping track of neighboring particles.

11.2 Lessons Learned

It's important to understand which tasks are cross-disciplinary and which tasks can be accomplished individually. When working with other people with different technical backgrounds, it's also important to be overly explicit with terminology to make sure everyone is on the same page.

Glossary of Acronyms

- ASIC - Application-specific Integrated Circuit
- BRAM - Block Random Access Memory
- CLB - Configurable Logic Block
- CPU - Central Processing Unit
- DRAM - Dynamic Random Access Memory
- FPGA – Field Programmable Gate Array
- GPU - Graphics Processing Unit
- HLS – High Level Synthesis
- LUTRAM - Lookup Table Random Access Memory

References

[1] M.Kroes Optimizing Memory Mapping for Dataflow Inference Accelerators: Efficient Memory Utilization on FPGAs. Delft University of Technology, 2020.

[2] M.Macklin and M. Müller. Position Based Fluids. ACM Transactions on Graphics, 2013.

[3] M.Martel Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. Form Methods Syst Des 35, 265–278 (2009). https://doi.org/10.1007/s10703-009-0068-y

[4] H.Su 3D Deep Learning on Point Cloud Representation (Analysis). Stanford CS468-17-Spring, 2017.

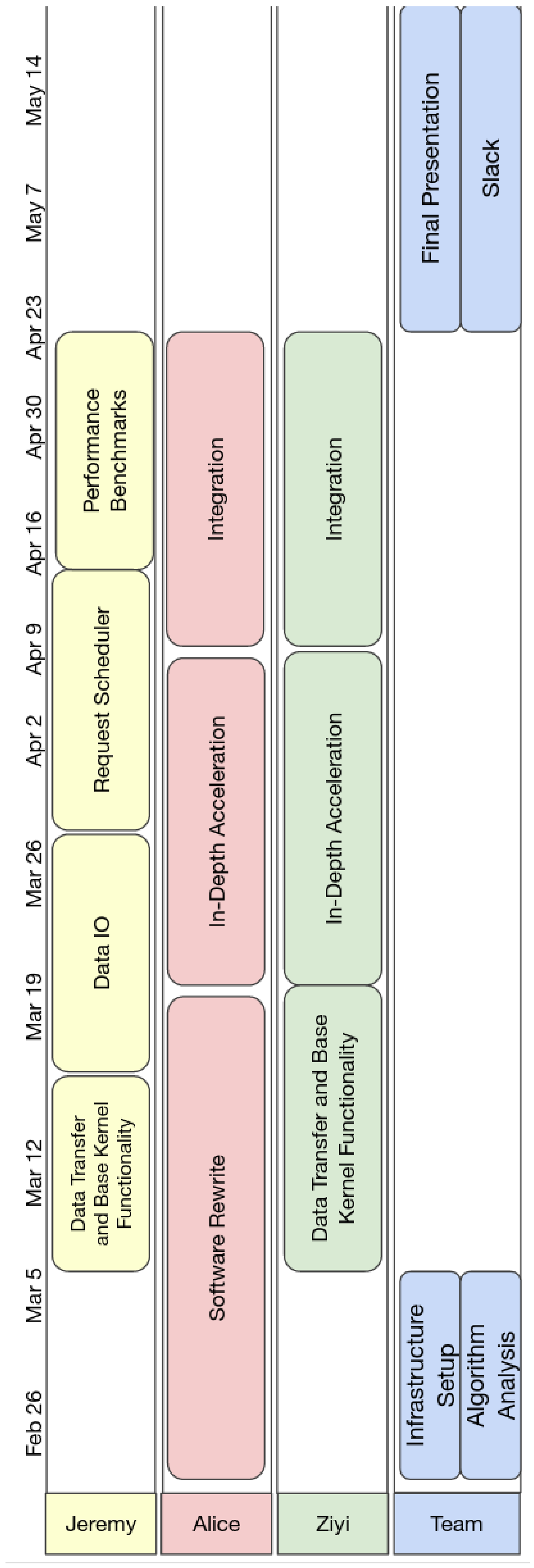


Figure 8: Gantt Chart