Water Works Final Report

Shonell Rowell
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
shonell@mit.edu

Isaac Taylor
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
iataylor@mit.edu

Abstract—We propose a 3D fluid simulation engine, which utilizes Smoothed Particle Hydrodynamics to produce visually realistic fluid simulations in real time. We aim to demonstrate that high-fidelity simulation can be accomplished by utilizing hardware's inherent capabilities for high-bandwidth parallelization, which can achieve greater performance than naive processor implementations.

Index Terms—Digital Systems, Field-Programmable Gate Arrays, Computer Graphics, Smoothed Particle Hydrodynamics

I. INTRODUCTION & MOTIVATION

Smoothed Particle Hydrodynamics (SPH) is a computational method used for simulating the mechanics of continuum media, such as fluid flow. The method works by dividing the fluid into discrete particles, and the ensemble of these particles emulates a continuous volume through interactions with each other. A scalar quantity A is interpolated at location $\mathbf{r}$ by a weighted sum of contributions from all particles [1]:

$$A_S(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h), \qquad (1)$$

where j iterates over all particles, and m_j, r_j, ρ_j, and A_j represent the mass, position, density, and specific property of the given particle. The function $W(\mathbf{r}, h)$ is a smoothing kernel with radius h, interpolating interaction strength with the distance between particles.

We simulate many particles and utilize this formulation to calculate essential quantities of the Navier-Stokes equations to produce pressure force vectors that cause the movement of our particles to evolve over time, exhibiting fluid-like properties. We then use our rendering engine to take the updated positions and produce an HDMI output to display them.

This process incorporates a lot of floating-point computation that is expensive on CPUs capable of only arbitrary bit manipulation, making it such that these operations are often performed on dedicated hardware modules. We aim to demonstrate how a dedicated design could efficiently implement the process.

II. SIMULATION ENGINE

The simulation engine sequentially computes all required terms for each SPH equation and then accumulates these density and force terms to update each particle's position and velocity properties by a single time-step. This is done for each frame as demanded by our rendering module's HDMI signal generator.

A. Binary16 Format

Our system utilizes the half-precision 16-bit floating-point format for computation. We chose to use floats for their expressivity over integers and the 16-bit variety to reduce memory footprint and avoid bandwidth limitations. This was done at the expense of the precision provided by full-width 32-bit floats, which later became apparent in the accuracy of the simulation. These modules serve as the foundation for the system's entire behavior, and their individual optimization contributes to the overall performance.

All arithmetic operations are performed through a suite of modules: binary16_adder, binary16_div, binary16_div_pipelined, binary16_multi, and binary16_sqrt. Each of these modules, excluding the singular 'div', is capable of receiving a new input on each cycle, making it possible to chain modules directly to one another to efficiently execute expressions containing multiple binary operators. The multiple versions of 'div' were constructed for scenarios where it is unnecessary to pipeline the computation and where utilization should be conserved. The modules were all individually tested by simulation on suites of random numbers to validate their accuracy.

Testing showed positive results for the modules. However, given that the binary16 format has reduced precision, errors over multiple operations can accumulate and drive our simulation further astray. Our implementation follows the IEEE 754-2008 standard in terms of the representation of the floats and their operations, but they lack specific rounding procedures that would help circumvent these issues. This would be a point of focus for future work. The multiplication module utilizes a single DSP48E1 to compute the multiplication of two mantissas in a single cycle. Care is taken to conserve resources by not unnecessarily instantiating these modules throughout the system.

The adder and multiplier each take 4 cycles, the dividers 24 cycles, and the square root, 13 cycles.

B. Particle Buffer

The Particle Buffer is an inferred BRAM module that holds position and velocity data for particles and serves as an interface between the simulation and rendering portions of the system. The BRAM module is inferred from an arbitrary maximum limit that we set on the number of particles that can be simulated at build time. At runtime, the number of particles the system should initialize and simulate is configurable, so we allocate enough space to allow for a range of options.

For each particle, we allocate 4 16-bit floats for its position and velocity in 2 dimensions, in a single address space. We chose to utilize a full 64-bit data width so that modules like the scheduler could access both properties of a single particle in a single cycle, enabling us to achieve maximum throughput on our term calculation pipeline. This comes at the cost of increasing register and BRAM usage, however.

The module has a simple two-port, two-cycle read interface. Our system began development with this choice in order to avoid complications with meeting timing before completion, and through pipelining, we are able to effectively nullify its waste. The buffer has its reads from port A output from the simulation module as it is used to transport information to the rendering module.

C. Scheduler

The scheduler is a simple FSM that iterates through particle pairs i, j, and assembles 'packets' containing the required variables for computing each of the terms in Equation 2 and 4. The scheduler utilizes the particle's current velocity and position from the buffer to predict its next position, and it uses that predicted position for further computation. This results in a more stable simulation, enabling greater stability than a standard forward Euler step.

The received data from the buffer is pipelined into a sequence of multiplication and addition binary16 modules, one for each dimension, that evaluate this predicted position. The scheduler sequentially reads from the buffer, issuing a different read every cycle, in order to achieve the maximum throughput for the subsequent computation module.

For density calculation, only the predicted positions are needed. But for force calculation, where pressure and density are required, the scheduler interfaces with the Accumulation module, which also contains another two-port, two-cycle read BRAM module serving 2 16-bit floats for a given particles pressure and density reciprocal. The scheduler thus synchronizes these reads so that the required information arrives at the same time, and a valid packet is sent downstream.

Once the scheduler has sent all the terms for the calculation of a single value, it stalls until it receives a signal from the accumulator expressing its completion, and then proceeds to update particle $i + 1$.

$$\rho_i = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h), \qquad (2)$$

$$p_i = k(\rho_i - \rho_0) \qquad (3)$$

$$\mathbf{f}_i^{\text{pressure}} = -m \sum_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h). \qquad (4)$$

D. Term Computation

The Term Computation module calculates both density and force terms, determining which to compute based on an indicator from the scheduler. It takes inputs of width $16 * 7 = 112$ bits. The input is made up of 7 16-bit floats that each refer to a different scalar value, in this format:

$$\{r_i, r_j, pressure_i, pressure_j, \frac{1}{\rho_j}\}$$

We choose to compute the reciprocal of the density and store that as opposed to the base value as it allows us to simply multiply this scalar value later on instead of divide by it, which is much less expensive. This would be wasteful in a scenario where one must divide each element of the force vector, vastly increasing utilization and delay. The module contains chains of arithmetic modules and buffers that enable it to achieve maximum throughput. It executes a computational graph of binary operations and comparisons that produces a scalar value or vector to be output to the accumulator.

This function of the module was originally planned to be exhibited by a suite of independent and unpipelined workers that would have inputs dispatched to them by a module which would observe their availability. We realized that not only would this be spatially inefficient, creating more expensive modules, but it would also be more complex to design an efficient arbitrator and protocol. This led us to fully pipeline each arithmetic module in order to achieve the simplest and optimal implementation in terms of cycle count.

E. Accumulation & Storage

The Accumulation & Storage module receives terms from the computation module and sums them. Upon completing the calculation of a density property, its reciprocal and the converted pressure values, as performed in Equation 3, are stored within BRAM. Here k refers to a configured pressure coefficient constant, and ρ_0, the target density of the fluid. We chose to instantiate a separate 2-port interface in this module, rather than adding it to the Particle Buffer, because it enables the Scheduler to access all necessary properties at once, without needing to expand the data width even further. It also avoids the need to design complex arbitration logic to decide which of the Updater, Update Buffer, or Scheduler modules gets to access any port at any specific time. However, it results in more memory being inferred than what might be strictly necessary for the module.

That covers the storage portion. The accumulation portion is handled by a suite of individual element accumulators, one for each dimension, that will continuously sum values they receive until signaled that no more are being generated by the computation module. Upon successful accumulation, the module signals the scheduler to resume calculation and move to the next main particle index $i + 1$.

Internally, each accumulator has a queue of length 3 for float values. Every cycle, when it holds at least 2 floats, it sends the pair to its single binary16_adder module. Every cycle during operation, the module can receive a new term from the computation module or a newly generated sum from the adder module, and there is logic in place to properly serve and store these values to ensure that no more than 3 values ever exist within the queue. Since you can serve as many terms as you receive each cycle, this is ensured. If there are no more terms in flight and the queue has only 1 element, the accumulator outputs its value to either be assembled into a force vector, undergo further processing, or be sent off to the Particle Updater.

F. Particle Updater & Buffer

The Particle Updater takes the complete force vector for a particle, reads out its position and velocity information, and computes the simulation step. It updates the velocity according to the acceleration dictated by the force and the particle's density, and then updates the position. Logic is written such that if a particle's new position places it outside the bounds of the simulation (as configured), it will negate the velocity in that direction. This ensures that particles remain within the simulation space and exhibit the container-filling property of fluids.

After these values are computed, they are sent to another buffer before going to the main particle buffer. This is to avoid overwriting the previous frame's data before all the computation is done. This buffer holds data until it is signaled by the scheduler that the frame is complete, at which point it overwrites the data within the Particle Buffer. We now realize that instead of instantiating new memory, the Particle Buffer could be used, and the scheduler could be configured to access a different address space every other frame, effectively double-buffering the simulation.

This completes the cycle of computation for a single frame of simulation. Depending on the number of particles present in the simulation, calculation of one frame will take different numbers of cycles. 16 particles takes 380 cycles per frame, 64 takes 22,000 cycles per frame, and 128 takes upwards of 1,000,000 cycles. When a frame doesn't complete in the allotted time for a single frame for a 30 fps output, the simulation will continue to work until it eventually completes, and frames may be skipped in the output. There is much room to further push timing on arithmetic operations, and try to de-synchronize processes within the system to gain some more speed up.

III. RENDERING ENGINE

The rendering engine consists of multiple rendering instances that draw the pixels of particles in parallel. These pixels are then accumulated in a frame buffer and rendered to HDMI 720p. The rendering pipeline includes a few components:

- **Perspective Projection**. The particle positions are moved from their world position into the screen space.
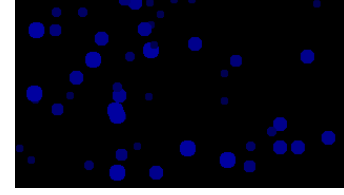

Fig. 1. Example frame of fluid simulation. Note the rendering is in 2D and does not include depth occlusion

Fig. 2. Particles being rendered to HDMI output display

- **Coordinate Normalization**. The screen space center positions are scaled to their pixel coordinates.
- **Rasterization**. The rendering engine draws each pixel of each spherical particle (which becomes a circle when projected onto the 2D screen).
- **Data movement.** The particle and pixel data need to be moved in and out of the parallel modules.

A. Particle Buffer Interface

The rendering engine is responsible for drawing each particle to the screen on each frame. To ensure that particles can be sent continuously, particle positions (x, y, z) from the particle buffer are first placed on a queue. The queue is attached to the module bus_driver.sv, which cycles the queue output to the parallel rendering instances. The cycling is controlled by a counter, which increments each time a new particle is sent through. The design currently uses 4 rendering instances. Each rendering instance draws the pixels for a single particle at a time.

B. Projector

The projector module is the beginning of the system's rendering pipeline. The projector transforms the particle center coordinates from world space to screen space to be used later by the rasterizer to fill in pixels that are covered by the particle. The projection is described by a Model-View-Projection (MVP) matrix, which allows one to specify world space transformations (model), world to camera space (view), and perspective projection in one 4x4 matrix. We choose this particular paradigm which is implemented by OpenGL, because it is standard and allows for intuitive normalization using homogenous coordinates.

The projector is an FSM with two stages (idle, projecting). The rasterizers follow a similar scheme. Even though the projection math is fully pipelined, because the downstream rasterization modules cannot be pipelined and take many cycles to complete, there would be unnecessary buildup. The
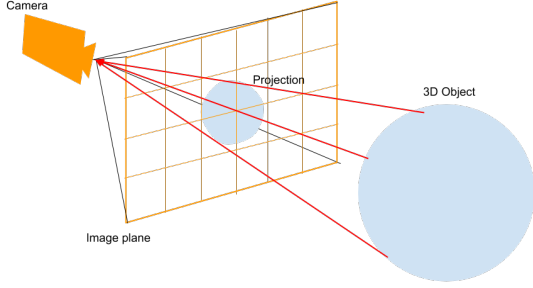
Fig. 4. Example Test Simulation for Rendering

Fig. 3. Rasterization Visualization

projector waits for the rasterizer to be ready before it moves into the idle state to be ready for new inputs.

For the projection itself, the position is multiplied by the 4x4 MVP matrix represented by 16 16-bit float parameters. The matrix multiplication and subsequent normalization occurs in stages which allows the computation to essentially vectorized over the four dimensions. First each column of the MVP matrix is multiplied by the position vector. Then two of the columns are added, and the vectors resulting from the addition are added together. The resulting homogenous coordinates are converted to normalized device coordinates (where all values lie between -1, 1) by dividing by the w coordinate. This now three-dimensional position (x,y + z depth) is scaled and rounded to screen coordinates. Each of these stages make use of the binary16 adder, multiplier and divider modules. At the end of the pipeline, the center screen coordinates and depth are available for the rasterizer.

C. Rasterizer

Similarly to the projector, the rasterizer is represented by a FSM with two states (idle, painting). When the projector sends a new particle center to the rasterizer, the painter FSM enters paint mode. The bounding box of the particle is calculated, and the painter loops through the pixels within that bounding box. Each pixel is checked against the circle formula ($x^2 + y^2 < radius^2$) to determine if it should be filled (currently the design uses a constant radius size regardless of depth). If the pixel is not within the circle, the painter moves to the next pixel in the bounding box. Each time a pixel is filled, the depth value and buffer address (based on pixel position) are sent to the frame buffer.

D. Pixel Manager

After rasterization, the pixel manager is receiving pixel positions that should be rendered on the screen. The pixel manager determines the color of the pixel which is solid blue then sends the color data to the frame buffer.

E. Arbitration

At this stage in the rendering pipeline, each of the parallel rendering instances are trying to write to the frame buffer. However, the frame buffer can only write one pixel at a time. In order to manage the multiple outputs of the parallel modules, we choose to create a module called arbiter.sv, which writes to the frame buffer one pixel at a time. The arbiter, similarly to the bus driver, is controlled by a counter, which cycles through the multiple inputs and pushes the data through if it is available. However, because rasterization can potentially have multiple rendering instances output at the same time, or have new pixels come in before the cycle wraps around (a problem which worsens when adding more parallel instances), we need queues to accumulate outputs from the rendering instances so that pixels are not dropped.

F. Frame Buffer

In order to accumulate the pixels to be read into HDMI later, we need a frame buffer to store pixel color data. In order to not write to a frame buffer being read, we use two frame buffers, one which is being read from and one to be written to. Once the global new frame signal is sent, the frame buffer roles switch. To keep buffer sizes low, we confine our screen coordinates to 320x180, and then upscale to 720p. With this size, we choose to use BRAM to store the data, which provides adequate space while minimizing the amount of data management required of a more complex memory like DRAM.

IV. EVALUATION

We evaluate our design based on its qualitative performance: stability, fluidity; along with its utilization. Upon observation, the simulation currently exhibits rather unstable execution. If too many particles are being simulated and create too many interactions, the simulation tends to explode as energy is continuously added to it. We assume that this is because of the errors that accumulate with more arithmetic operations, and because the modules lack proper rounding, this effect is exacerbated. We also believe that further tinkering of simulation parameters could improve stability as well.

Our final design demonstrated in our project video only achieved 30.31% slice utilization and 79.33% BRAM utilization. For further work, more logic could be dedicated to adding more computation modules to increasing throughput that way, effectively halving execution regardless of particle count. We used only 23 DSP48E1 modules, only being consumed by binary16 multiplier's.

Our minimum viable product was to be our 2-D simulation, and our system was able to produce something in its likeness, but it lacks the quality and robustness that was sought. Complications in the simulation's development due to redesigns and bugs within float arithmetic led to delays. The overall complexity of this mvp was underestimated, a more

succinct initial design would have made things easier later on, but many optimizations where invisible until they were encountered directly.

We were not able to fully incorporate the 3-D rendering pipeline into our system, as it wasn't completely polished on its own and there were complications in integration. If things were to be done over, we would have set more checkpoints for dual progress between our two parts. We thought that it would be easier to have things be separated and worked on in parallel, but in order for that strategy to be effective, ironically, more collaboration must be done.

V. AUTHOR CONTRIBUTIONS

Shonell Rowell handled all of the float arithmetic and simulation pipeline while Isaac Taylor handled the rendering. We both worked on the preliminary report, presentation, final report, and project video.

REFERENCES

[1] Müller, Matthias & Charypar, David & Gross, Markus. (2003). Particle-Based Fluid Simulation for Interactive Applications. Fluid Dynamics. 2003. 154-159.

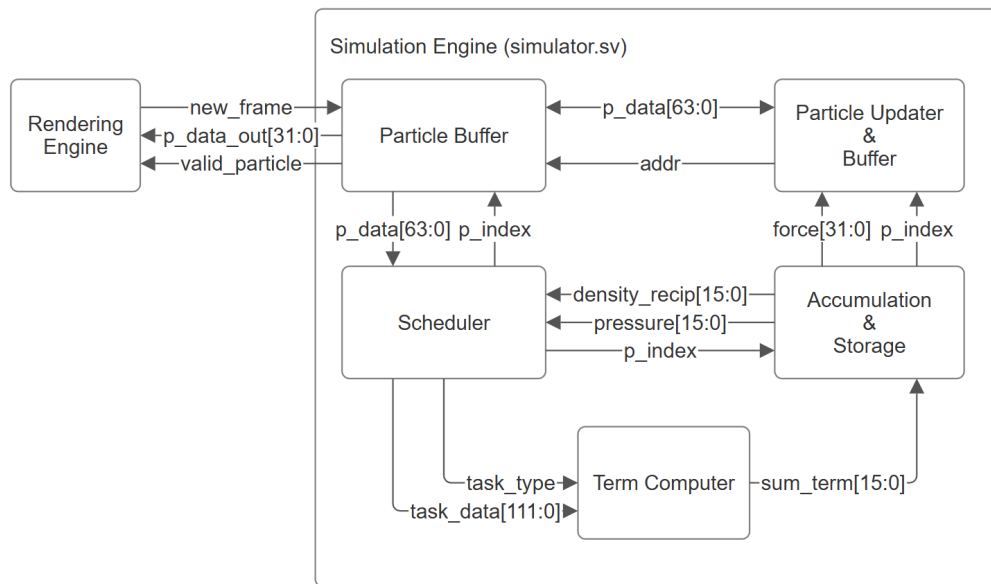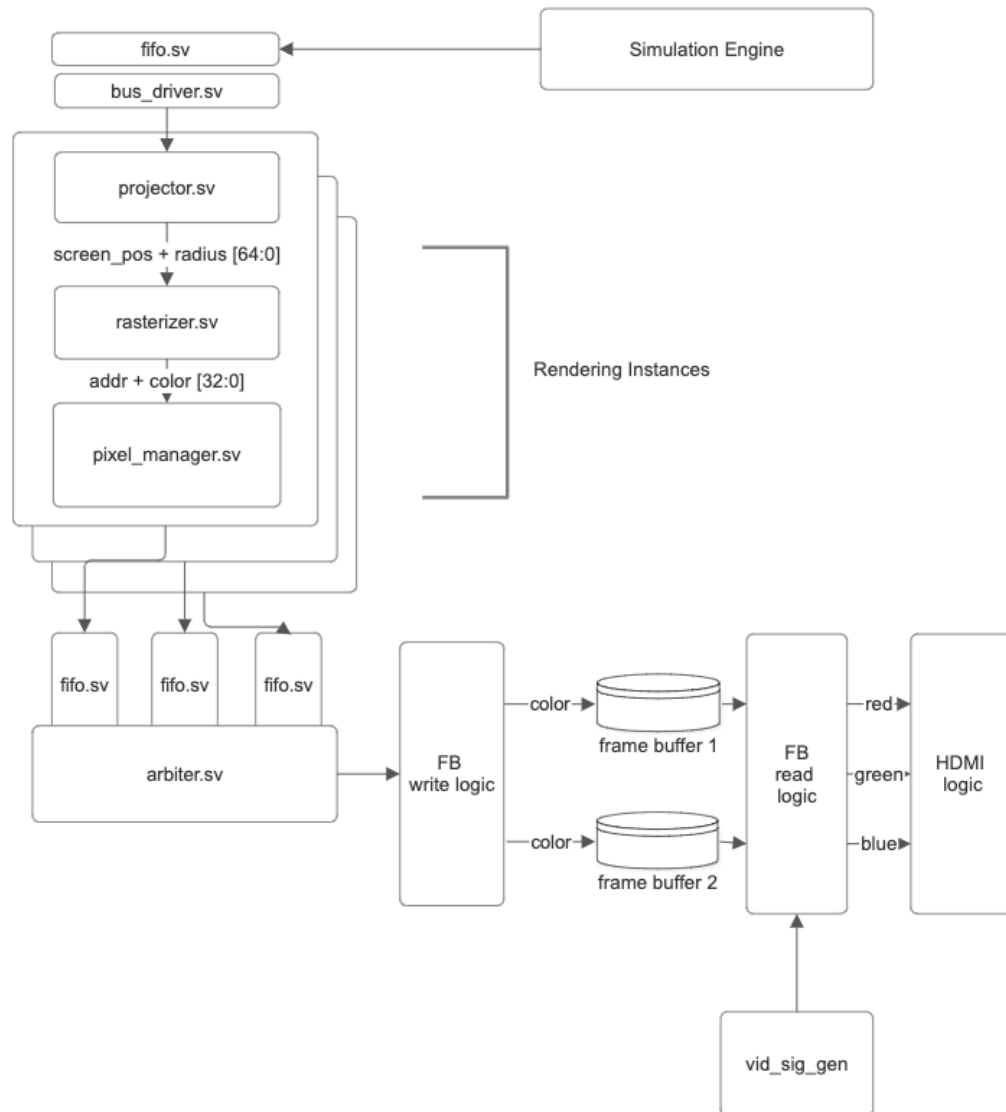[2] Source Code: https://github.com/ChanellR/6.205-Project

Fig. 5. Block Diagram for Simulation Engine

Fig. 6. Block Diagram for Rendering Engine