EE2 Mathematics Accelerator

Charlotte Gibson, Jeremy Tan, Nishant Kidangan Mathew,
Lily Martin, Kayvan Faghani

June 2025

GitHub Repository

Contents

1 Abstract

This report details the design and implementation of a real-time, FPGA-accelerated Computational Fluid Dynamics simulation and visualisation system. The system visualises the evolution of a fluid domain using the Lattice Boltzmann Method, selected for its inherent suitability for embarrassingly parallel execution and efficient implementation on hardware.

To achieve performance targets, the Lattice Boltzman solver is implemented on the PYNQ-Z1 FPGA, utilising a modular and parameterised architecture that supports scalable parallelism constrained only by the FPGA's available resources. A dual-BRAM scheme ensures safe streaming and collision operations per lattice node, while a pipelined collider design maximises throughput under strict resource constraints.

A custom cache and DMA-based streaming system enables real-time data transfer between the FPGA and external DDR memory, supporting higher-resolution simulations while maintaining a minimal target computational cost. The FPGA exclusively handles simulation, while a host-based frontend implemented in Unity renders high-quality visualisations at 60 FPS and provides an intuitive user interface for real-time parameter control and educational exploration.

To facilitate user interaction, a Python-based drawing application allows custom barriers to be designed and inserted dynamically into the simulation domain.

The system architecture employs robust TCP-based networking for low-latency frame delivery, decoupling computation and rendering while ensuring consistent real-time performance.

The final implementation demonstrates significant acceleration over CPU-only approaches and delivers an engaging, responsive educational tool for visualising fluid dynamics concepts.

2 Project Proposal

2.1 Initial Design

The initial concept for the project was driven by an interest in real-time dynamic fluid simulation. In this initial design the FPGA platform was tasked with performing the 3D fluid simulation computations, while the rendering and user interface was delegated to a host system running a Unity-based application. The Unity frontend would provide a rich, interactive environment, allowing users to influence the simulation in real-time through intuitive UI elements.

However, an early feasibility study highlighted significant resource constraints associated with implementing a 3D simulation on the available FPGA hardware. Detailed analysis showed that 3D simulation would quickly saturate the FPGA's on-chip memory, severely limiting the achievable resolution and domain size. It was determined that more advanced memory management strategies could potentially enable a functional 3D implementation, developing and optimising this for a 3D simulation was not feasible within the scope and timeline of the project. Despite this limitation, the team remained committed to the vision of creating a dynamic fluid simulation. To align with practical constraints, the project was refined to focus on a high-quality 2D fluid simulation.

2.2 Final Design

The final project design refines the original vision into a high-performance, technically feasible 2D fluid simulation with a strong focues on interactivity and educational value. In this configuraiton, the FPGA handles all of the fluid dynamics calculation. The simulation algorithm is optimised to fit with the FPGA's available resources, ensuring real-time behaviour.

In this final configuration, the FPGA is dedicated to performing the real-time numerical fluid dynamics computation. The FPGA communicates simulation results to the host which uses Unity to render the fluid flow dynamically and provide a polished, interactice user interface.

A key enhancement in the final design is the development of a custom drawing application embedded seamlessly within the Unity environment. This tool allows users to intuitively draw and edit barriers within the fluid domain. This interactivity transforms the simulation from a passive visualisation into an active, exploratory learning tool.

Overall, the final design successfully achieves the project's objectives by showcasing the performance benefits of FPGA-accelerated computation, delivering an engaging and highly interactive simulation environment, serving as a comprehensive educational tool for exploring the fundamentals of fluid dynamics in real-time.

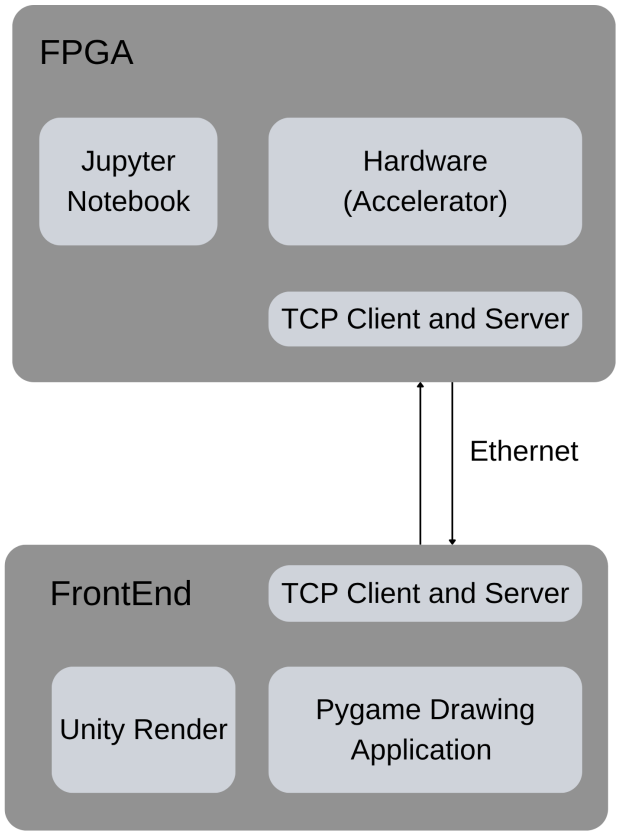

Figure 1: Final Design Architecture

2.3 Project Management

Effective project management was essential to the successful delivery of this project. From the outset, the team prioritised regular communication, clear task allocation, and planning to ensure efficient progress towards an early minimum viable product (MVP) which was then iteratively enhanced.

Regular collaboration was supported by a structured meeting schedule, which helped maintain momentum and enabled rapid problem-solving. The team met in person at least three times a week—on Mondays, Tuesdays, and Thursdays—and increased the frequency of meetings during the final integration phase, which required more intensive in-person work. These meetings were complemented by ongoing communication through group messaging and calls, ensuring all team members stayed up to date with progress.

A private Github repository was established for version control and collaborative development.

2.3.1 Task Allocation

At the outset of the project, tasks were assigned based on each member's interests and expertise to maximise productivity and ensure a high-quality outcome.

Name	**Role**
Nishant	FPGA implementation
Lily	Drawing software and cache system
Jeremy	Collider, LBM solver debugging and testbenching
Kayvan	LBM solver development
Charlotte	Unity rendering, frontend, and cache system

While primary responsibilities were clearly defined, members provided support across tasks particularly during integration. Debugging, brainstorming, and performance tuning were highly collaborative, contributing to a final system satisfying all pre-defined requirements.

2.3.2 Sprints

An Agile development methodology, structured around focused sprints was employed to manage the timeline effectively. Each sprint had clear, achievable goals that progressed the project towards a fully integrated and tested system. Sprint plans were adjusted as needed to accommodate technical challenges or rapid progress.

Prioritising early delivery of a functional MVP enabled continuous testing and integration This iterative cycle of refinement guided by sprint reviews progressively improved system performance.

Sprint	Date	Objective
Sprint 1	19/05	• Research CFD algorithms suitable for project brief. • Prepare project proposal for staff review. • Set-up Vivado and Pynq for FPGA development.
Sprint 2	22/05	• Design and develop a prototype LBM simulation in Python. • Design data transfer architecture. • Develop an initial Unity frontend for basic real-time visualisation. • Design and begin implementation of the drawing software interface.
Sprint 3	27/05	• Implement Verilog LBM solver and collider. •Initial integration with FPGA. • Implement additional render modes in Unity. • Integrate Unity frontend with TCP data stream. • Extend drawing application feature set. • Prepare and deliver interim presentation.
Sprint 4	03/06	• Validate the single-cycle LBM solver. • Integrate validated components into MVP. • Pipeline and parallelise the LBM solver and collider. • Optimise Unity communication and rendering pipeline. • Implement and integrate interactive Unity UI. • Research and design the cache system for higher-resolution simulations.
Sprint 5	09/06	• Implement and integrate the cache system. • Full system testing and optimisation. • Report writing and delivery. • Demo preparation and delivery.

3 Project Specifications

The following fundamental requirements are explicitly defined in the project brief:

1. Display visualisation of a mathematical function in real-time.
2. Perform embarrassingly parallel computation on and accelerated by an FPGA.
3. Provide a user interface to enhance the visualisation as an educational tool.

To fulfil the brief for the selected Computational Fluid Dynamics (CFD) application and produce a high-quality project, the following self-imposed targets have been defined:

1. Fluid Simulation Requirements
   (a) Employ a CFD algorithm inherently suited for parallel computation.
   (b) Demonstrate measurable benefits of FPGA acceleration compared to a CPU by achieving:
      i. A target computational cost of at most 40 clock cycles per lattice cell at full parallelisation.
      ii. A simulation grid resolution of 300×100 with provision for higher resolutions.
   (c) Design a parameterised and modular system to support scalable parallelisation.
   (d) Support simulation of arbitrarily large domains, constrained only by FPGA memory capacity.
2. Frontend Requirements
   (a) Achieve a target rendering rate of 60 FPS to maintain real-time visualisation.
   (b) Support real-time visualisation of multiple fluid properties.
   (c) Provide a responsive, real-time control interface for adjusting simulation parameters during execution.
3. UI Requirements
   (a) Allow users to draw and insert custom barriers in the simulation domain.
   (b) Support interactive barrier manipulation, including placing, repositioning, and resizing.
   (c) Implement functionality to save and load user-created barrier configurations for repeatable experiments.

4 Design Process

4.1 CFD Algorithm Choice [2]

CFD problems are generally approached using either Lagrangian (particle-based) or Eulerian (grid-based) methods, both of which numerically solve the Navier–Stokes (NS) equations.

For the Lagrangian approach, methods such as Smooth Particle Hydrodynamics are commonly used. However, these particle-based methods are not well-suited for parallel computation, as they require interpolation between nearby particles. This typically involves $\mathcal{O}(n^2)$ time complexity for neighbour searching and results in irregular memory access patterns.

In contrast, the Eulerian approach employs a fixed grid, enabling regular memory access, thus making it more suitable for parallel computation. Traditional methods in this category include the Finite Difference Method, Finite Volume Method, and Finite Element Method. However, for this project the Lattice Boltzmann Method

(LBM) was selected, as it aligns most closely with the project requirements, offering significant advantages in terms of parallelisability and implementation efficiency.

LBM does not require direct solving of partial differential equations, which is a key advantage over traditional methods that require explicit solutions to the NS equations. Directly solving the NS equations requires solving pressure Poisson equations and performing gradient or divergence operations, which are computationally intensive and challenging to parallelise. By avoiding these steps, LBM employs the simpler lattice Boltzmann equations (LBE), yielding a second key advantage: excellent parallelisability. The most computationally demanding tasks, streaming and collision, are performed locally at each lattice cell. This locality allows operations to be executed independently across multiple cores or threads.

Despite these benefits, one disadvantage of LBM is its high memory usage. Streaming distribution functions across the lattice involves frequent memory access operations, which can become a performance bottleneck in memory-constrained systems.

4.2 LBM Algorithm Explanation

LBM operates at a mesoscopic scale, where the behaviour of virtual particles is tracked rather than solving the fluid equations directly. The LBM process is defined by two main stages: *streaming* and *collision*. In this case the D2Q9 model is used, which refers to two spatial dimensions and nine discrete velocity directions. These include eight outward-pointing directions and one stationary point at the centre of each lattice node.

During the streaming stage, particle distribution functions are propagated to neighbouring lattice nodes along the defined velocity directions. In the subsequent collision stage, the distributions at each node are locally relaxed toward an equilibrium state.

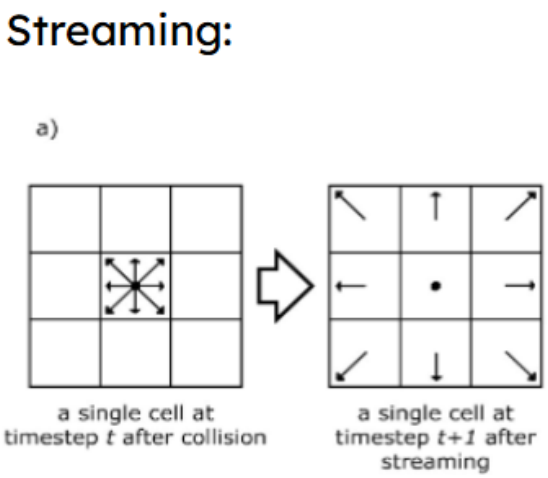

Figure 2: LBM Streaming Stage

This behaviour is described by the LBE with the Bhatnagar–Gross–Krook (BGK) approximation. It governs the update of the distribution function f_i, which represents the number of particles moving in direction $\vec{e}_i$ at position $\vec{x}$ and time t. An equilibrium distribution function is also applied to model the relaxation process toward fluid equilibrium.

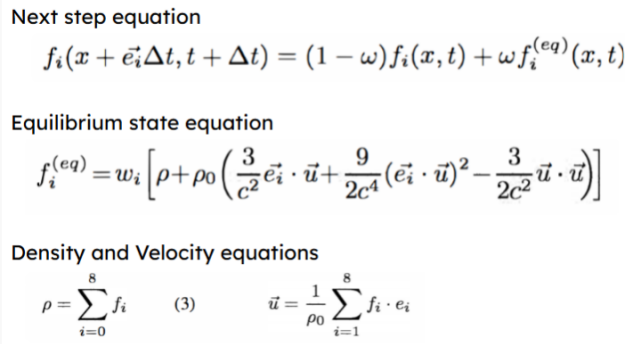

Figure 3: Collision Equations

4.3 Solver Architecture

High-Level Design

In order to implement the LBM algorithm we created an LBM solver module. The solver is connected to 18 BRAMs (2 for each velocity direction) and the collider module discussed in Section 2.4. The solver itself contains a state machine controlling the read and write addresses, input data, and write enables of all 18 BRAMs. The solver takes an additional n-bit input wire called barriers, where n is the number of simulation cells. A cell's barrier status is represented by a 1 at its given index if a barrier is present, otherwise a 0. This enables users to simulate fluid flow around objects by using barrier cells, which reflect incoming values. Each cell outputs four quantities: x-velocity (ux), y-velocity (uy), velocity magnitude squared ($|u|^2$), and density (ρ).

BRAMs

Each of the 9 velocity directions was assigned a dedicated BRAM, since for all algorithm stages except collide, the value and destination of each vector can be independently computed and written. A second BRAM was added for each velocity direction to prevent overwriting of current BRAM values. The second BRAM allows the current state of the cell and the incoming velocities to be considered together to determine the correct next state of a cell. Without this safety mechanism initial values would propagate to the boundaries and dominate the simulation.

Solver Algorithm

A state machine was implemented to control the BRAM inputs. This approach naturally fits the two LBM stages - stream and collide. Additional FSM stages were required to handle the simulation of barriers, and the zero-gradient boundary condition.

Each stage has designated states, described below, with implementation details discussed in Section 1.1.

- **Stream**: For each non-barrier cell, each of its directions is written to the adjacent cell being pointed to. For example, the East direction of a given cell is written into the East direction BRAM for the next time step. (Figure 4).

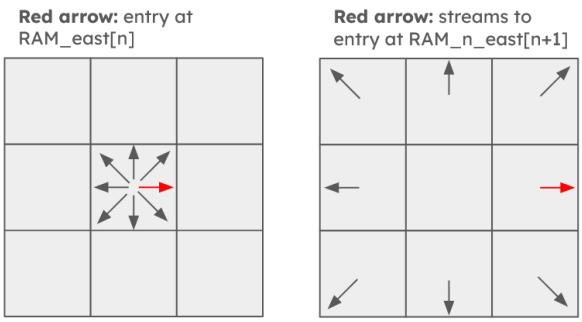

Figure 4: Streaming Illustration

- **Boundary**: To ensure the simulation is physically accurate, boundaries are handled using the Zero-gradient boundary condition. This extrapolates the fluid velocities to the boundaries, allowing the simulated fluid to exit the domain freely. To achieve this, the lattice's outer margin assumes the same inward-facing velocity as the inner margin (Figure 5).

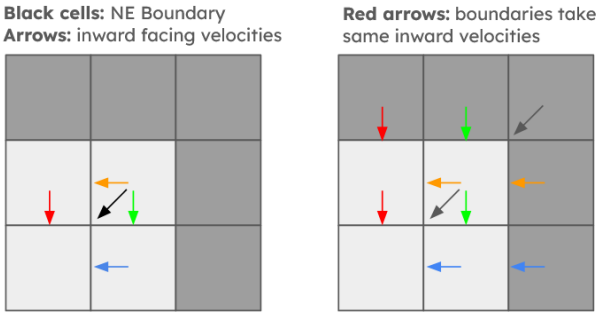

Figure 5: Boundary Illustration

- **Bounce**: For each barrier cell, incoming vectors during the streaming stage are reflected back along their incoming paths. For example, if an E vector is streamed into a barrier cell, its value is reflected W into the adjacent cell to the left (Figure 6).

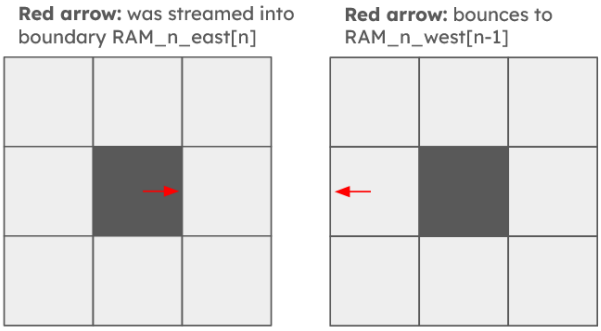

Figure 6: Bounce Illustration

- **Collide**: For each non-barrier cell the equilibrium values (LBE) and the BRAM values for the next timestep are computed, and the current timestep BRAM is written with the new values. ρ, $|v|^2$, ux, and uy are also written at this stage (Figure 7).

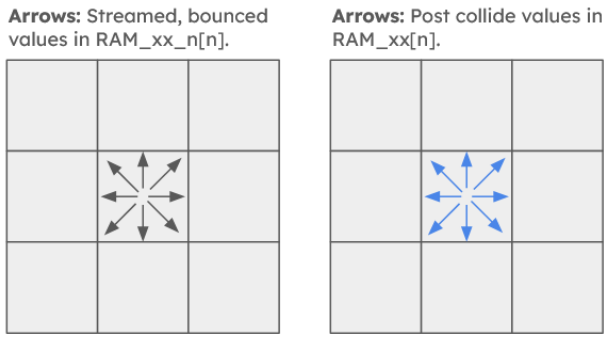

Figure 7: Collision Illustration

Solver Parallelisation

After validating the initial single-cycle LBM solver, development proceeded to a parallelized version. The parallel LBM solver enables embarrassingly parallel computation of lattice cells, significantly reducing per-frame compute time. The bit width of the BRAM was increased to enable reading and writing of data from multiple contiguous address at once. To achieve this, the LBM solver was divided into two parts:

- An LBM controller which manages the write addresses, indices, and the module states.
- An LBM parallel solver which computes the write enables and data input for each cell. This modular design was selected for its scalability: a single LBM controller is able to coordinate an arbitrary number of parallel solvers, allowing cell computations to scale with available resources.

As discussed, the LBM controller is linked to an arbitrary number of LBM parallel solvers. The controller connects directly to the BRAMs, determining the data to be written by evaluating the computed inputs for each cell against their respective write signals.

The LBM parallel solver module is controlled by the LBM controller. It receives the BRAM read address and data for a cell, computes the corresponding output data, and generates the write enable signal for that cell. This design enables scaling by replicating solvers up to the FPGA's capacity, reducing input bandwidth per solver by a factor of n, where n is the number of solver. Each solver includes it's own collider ensuring fully parallel computation.

4.4 Collider Design

Fixed-point vs Floating Point

Fixed-point arithmetic was chosen over floating-point arithmetic for a number of reasons. The most important benefit is that fixed-point operations are completed in a fixed number of clock cycles, which simplifies pipelining and ensures predictable performance. Additionally, fixed-point arithmetic is lower latency than floating-point and eliminates the need for dedicated floating-point unit reducing hardware complexity.

Q3.13

A Q3.13 representation was selected for calculations based on LBE research , which revealed that typically only 3 significant integer bits are required to accurately represent the fluid velocity and density values [1]. A 16-bit word length was deemed sufficient to maintain numerical stability, simulation accuracy, and

low memory usage. A 32-bit format was considered but ultimately rejected, due to significantly increased memory demands without providing meaningful improvements in resolution or rendering quality. Given the memory-intensive nature of LBM, memory efficiency was prioritized during system design. To summarise, the Q3.13 format allocates 3 bits for the integer part (including the sign bit) and 13 bits for the fractional part, using 16-bit signed integers.

Handling Multiplication

Multiplication [edit]

```
// precomputed value:
#define K   (1 << (Q - 1))

// saturate to range of int16_t
int16_t sat16(int32_t x)
{
    if (x > 0x7FFF) return 0x7FFF;
    else if (x < -0x8000) return -0x8000;
    else return (int16_t)x;
}

int16_t q_mul(int16_t a, int16_t b)
{
    int16_t result;
    int32_t temp;

    temp = (int32_t)a * (int32_t)b; // result type is operand's type
    // Rounding; mid values are rounded up
    temp += K;
    // Correct by dividing by base and saturate result
    result = sat16(temp >> Q);

    return result;
}
```

Figure 8: 16 bit Q format multiplication

To implement signed fixed-point multiplication in hardware using the Q3.13 format, a C-style algorithm was used as reference to write the hardware code.

The multiplication is implemented according to the following steps:

1. Convert both operands to `int32_t` to prevent overflow.
2. Multiply the two values.
3. Apply rounding by adding $2^{12} = 4096$ before the right shift.
4. Perform a right shift by 13 bits to scale the result back.
5. Apply saturation to ensure the result stays within the 16-bit signed integer range.

This method ensures that rounding and clipping are correctly handled, making it suitable for hardware implementation in an FPGA.

Handling Division

LBM requires division to compute the macroscopic velocity, specifically $\vec{u} = \sum(f_i \cdot \vec{e}_i)/\rho$. Due to the high computational cost of division on hardware without a dedicated divider, alternative algorithms were explored.

A lookup table (LUT) approach was considered but rejected due to excessive memory requirements. Representing the full dynamic range of ρ requires a large number of entries, exceeding the FPGA's available

resources. Traditional division algorithms, such as *radix-2* and *high-radix* division, were also evaluated using Vivado's IP core offerings. However, both approaches were deemed unsuitable for high-throughput systems due to high latency and incompatibility with pipelining.

To overcome these limitations, the Newton-Raphson method was adopted to approximate $1/\rho$, enabling the costly division to be replaced with multiplication—a far more efficient operation in fixed-point arithmetic. Three iterations of the Newton-Raphson update were performed, to provide sufficient accuracy for the simulation. Since ρ typically remains close to 1 in the domain, an initial guess of $x_0 = 1$ was used. This choice encouraged rapid convergence and resulted in a high-quality approximation of $1/\rho$ with minimal hardware overhead.

Handling Square Root

During the collision stage, the squared velocity magnitude ($|\vec{u}|^2$) is computed. The initial plan was to compute the square root of this value in hardware before passing it to the front end for rendering. However, implementing the square root on the FPGA was found to be too resource-intensive due to high DSP slice consumption and increased pipelining complexity.

For optimal FPGA-resource usage, the square root operation was omitted, and the squared velocity was instead transmitted directly to the front end. The frontend system can easily compute the square root leveraging GPU where resources are less constrained and the performance impact is minimal.

Collider Pipelining

Two key reasons lead to pipelining of the collider design.

1. The collider contains significant combinational logic due to LBE computation resulting in a long critical path. Pipelining divides this logic into smaller stages, allowing the FPGA to be clocked at a higher frequency.
2. The collider is the most resource-intensive module in the design, so duplicating it per processing block would be highly inefficient. Pipelining enables a single collider instance to process multiple values sequentially, achieving high throughput with minimal resource overhead.

Additionally, since the collider is the most resource-intensive module, duplicating it per processing block is impractical. Pipelining enables a single collider instance to process multiple values sequentially, achieving high throughput with minimal resource overhead.

4.5 Cache System

To upscale the resolution of the fluid simulation from 50×50 to the target 300×100, a dedicated caching mechanism has been implemented to handle the large volume of data transfers between on-chip BRAM and external DDR memory. The overall resolution/lattice size of the simulation is thus only constrained by the total space available on DDR memory (512MB). A high-level illustration of the operation of the cache system is shown in Figure 9.

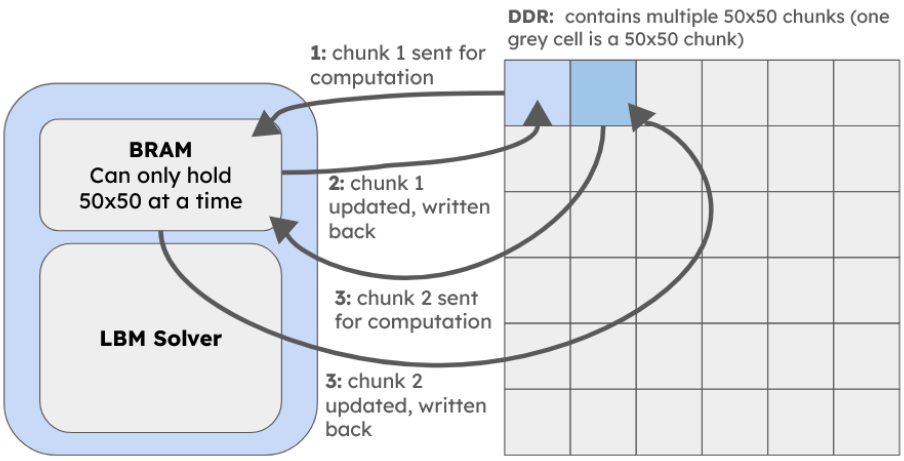

Figure 9: Cache system at a high level

The architecture introduces a buffering layer between the LBM solver and the DDR memory, handled through an AXI DMA interface. The cache system is divided into two custom hardware modules:

- A BRAM controller responsible for streaming solver output to the DDR.
- A DDR writer that retrieves stored data and repopulates the BRAMs in preparation for the next computational step.

Each of these modules operates according to the ARM AXI streaming protocol. To maintain data integrity a coordination mechanism ensures the BRAM writer does not overwrite memory locations needed by the reader. This approach is able to maintain a continuous flow of data at higher resolutions while adhering to the strict bandwidth and resource constraints of the FPGA.

A `frame_ready` signal acts as the principal trigger for a new transfer signal signalling that the LBM solver has produced a complete frame and that it is safe to commit to DDR. Once signalled, the controller and writer iterate over the frame in a predictable, address-sequential manner, resetting at the start of each new frame. The AXI DMA is controlled through the Jupyter notebook to continually allow this transmission of data.

Having two states for both IPs reduces the cycles required to transmit the data between the DDR and BRAMs, similarly to subsubsection 4.7.1.

4.6 FPGA-Specific Design

Initial Design and Integration with Xilinx IP

Due to the design choice to offload all rendering to the host (justified in Section 2.8.1), the FPGA is used exclusively for fluid simulation computations, hence the overlay was designed without using the provided template that included rendering and peripherals. First, research was undertaken to determine how to connect the custom Verilog RTL modules externally. After reviewing the Pynq documentation, AxiGPIO blocks were chosen as the interface to enable I/O port read and write operations for the RTL module. These AxiGPIO blocks can be written to and read from the Pynq's Jupyter notebook, making implementation straightforward. Utilising a premade AXI IP ensured correctness and proper integration with the Zynq PS and Xilinx IP. An example of this in the final design is the `param_setter` module Figure 10, which was used to pass in simulation parameters to the main solver.

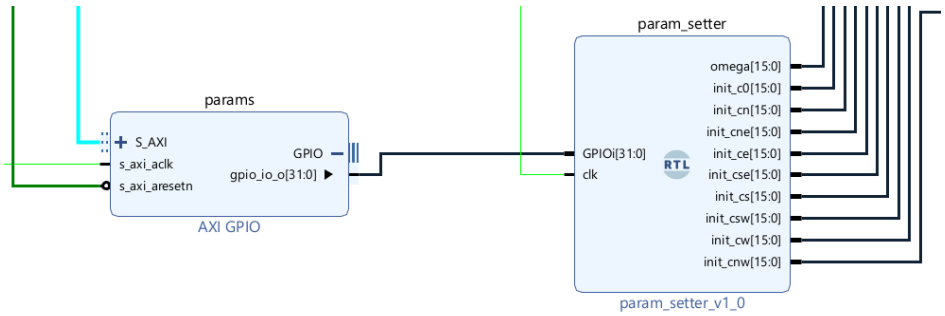

Figure 10: `param_setter` module interface with AxiGPIO

Issues with AxiGPIO

Despite the simplicity of this approach, two primary issued were encountered during development. Firstly, while Vivado is able to synthesis a design with numerous AxiGPIO, the Pynq python software is only able to recognise up to 8. Beyond this, the AxiGPIO are not visible in the overlay hierarchy, making them inaccessible. This is bug is not currently documented by Pynq so could not have been foreseen. The second issue faced, was high latency data-transfer. Combined, these issues resulted in later design iterations moving away from AxiGPIO.

Using BRAM

As previously stated, LBM requires storage of 9 velocities over two timesteps per lattice cell. For a lattice of size of 50 by 50, this results in $50 \times 50 \times 9 \times 2 \times 16$ bits $= 87.9$KB of data. Due to high frequency read and write operations required by the system, low-latency storage was essential. This lead to use of Zynq BRAM with total capacity 630KB and 1 cycle read latency.

Collider

As discussed in Section 2.4, the collider performs the majority of the LBM algorithm computation, so system performance measures such as maximum clock frequency and cycles per cell were tied to its efficiency Zynq's DSP slices were leveraged to perform multiplication and multiplication sums.

4.7 System Interfaces

Two main interfaces were designed to connect the overlay with the rest of the system. The first interface exposes programmable logic (PL) calculation results to the processing system (PS). The second interface exposed the PS data (from the PL) over a network interface to the host rendering application (Figure 41).

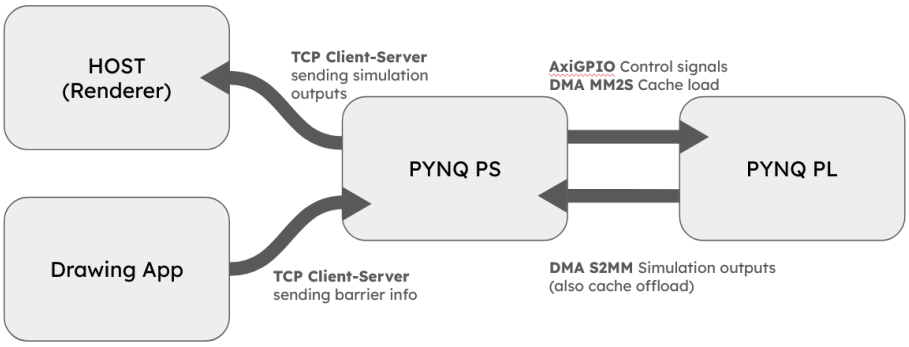

Figure 11: Interface Diagram

The design of two separate interfaces for this link rather than a single ethernet cable was a motivated by optimising the PL hardware for simulation computation rather than ethernet drivers. This is consistent with one of the project's core design principles - to focus FPGA resources solely for computation rather than rendering. Additionaly, the PS software is already capable of interfacing in this way so there was no benefit to using ethernet.

4.7.1 PL-PS Interface

This interface is responsible for buffering the per-pixel data outputs $u_x, u_y, \rho, |u|^2$ from the collider and making them available to the PS. Buffering is required because the collider only outputs sensible output values in the collision stage for 1-2 cycles.

Initial Design

The initial interface design utilised a query-based system, where the module consisted of two high-level states: `DATA_FILL`, and `QUERY`. This design was reliable, however the per-pixel AxiGPIO read and write incurred a high latency cost and became a major bottleneck to the simulation's render rate.

Final Design

Direct memory access (DMA) was then researched as an alternative approach to improve data-transfer latency. This is a technique whereby certain parts of the PL are accessed as part of the memory map, allowing for direct access. Vivado has a built-in AxiDMA IP which utilises a high performance slave port (HP0) on the Zynq processing system to transfer data at a high bandwidth. The Pynq also includes a Python API to interface with this block, such as the `axi_dma.recvchannel.transfer()` function and others in the class. An example of its use can be seen in snippet Figure 12.

```
for step in range(100, int(1e6), 100):
    write_step(step) # simulate until this step
    time.sleep(0.1) # wait for sim
    print(f"current step: {ostep_countn.read()}")
    dma.transfer(frame_buffer)
    dma.wait()
    frame_buffer.invalidate()
```

Figure 12: DMA Python Snippet

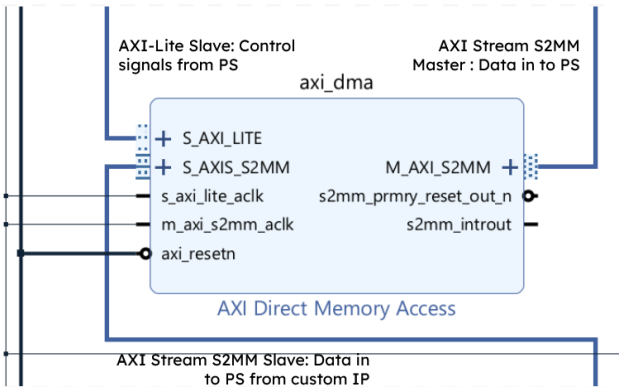

Figure 13: AXI DMA Block

The DMA block (Figure 13) can be setup to read from PL memory via the S2MM (stream to memory-mapped) AXI-Streaming interface. Such an interface will stream out data which is loaded into the PS memory (likely in DRAM). Researching DMA gave us the idea to use it as a caching system for our solver, which will be discussed later.

A custom AXI-streaming IP was implemented to use the DMA, with the design following the ARM AXI-Streaming protocol specification [3]. Inspiration from the pervious approach was taken with regards to the buffering of collider results. The interface is implemented as a Mealy-FSM with 2 states. Care was required to ensure that the FSM state and output transitions aligned with those in the specification. To guarantee timing alignment, the design was simulated with a variety of testcases in Verilator until results aligned with the protocol specification.

4.7.2 PS-Host Interface

Data transfer from the PS to the host renderer was performed using TCP over Ethernet. Ethernet bandwidth accommodates the full transfer of a simulation grid within the time constraints for real-time rendering. Testing with dummy data (Figure 14) yielded an average transfer rate of 126FPS for lattice dimensions of 300 by 100 , which comfortably meets the target design specification. Sufficient channel bandwidth reduces the delivery of each frame to a single `sendall()` operation, avoiding fragmentation and minimising latency. TCP guarantees reliable and ordered data transfer, which is essential to prevent visual artifacts. UDP was dismissed due to its lack of delivery guarantees. Mature TCP libraries available in both C# (Host platform) and Python (Pynq Jupyter Notebook) minimise potential integration issues and reduce development time.

4.8 Frontend Renderer

4.8.1 Rendering Approach

Two rendering approaches were evaluated for the fluid dynamics simulation.

The first approach offloads rendering responsibilities to a host platform. The FPGA exclusively performs numerical calculations for the simulation.

```
In [3]:
# Pynq code
import socket
import time
import numpy as np

host_ip = "192.168.2.1"
data_size = 4.5*2**20 // 8 # 4.5MBits
data = np.random.bytes(data_size)
```

```
In [16]:
avg = 0

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((host_ip, 8000))
    for i in range(1000):
        start_time = time.time()
        try:
            s.sendall(data)
        except:
            continue
        elapsed = time.time() - start_time
        avg += elapsed
#         print(f"{elapsed:.2f} seconds")
avg /= 1000
print(f"avg: {avg}")
print(f"max. possible FPS: {1/avg}")
```

```
avg: 0.007902304172515869
max. possible FPS: 126.54536932126577
```

Figure 14: TCP test with dummy data

Advantages	**Disadvantages**
• Supports design of advanced user-driven interfaces with interactive elements. • Host GPU enables advanced rendering techniques, significantly improving visualisation quality. • FPGA resources are dedicated to numerical computation, optimising the simulation resolution.	• Transfer of data between the FPGA and host incurs latency.

Table 1: Advantages and disadvantages of offloading rendering to a host platform.

The alternative approach was FPGA-based rendering. Both numerical simulation and pixel data generation occur on the FPGA. The pixel data is stored within BRAM and output through HDMI to an LCD panel.

Advantages	**Disadvantages**
• Eliminates data transfer latency.	• Competition between rendering and simulation logic for BRAM, the limiting resource. • Minimal capability for dynamic visuals and interactivity.

Table 2: Advantages and disadvantages of rendering directly on the FPGA.

Host-based rendering was selected to optimise visual quality and meet UI flexibility and interactivity requirements. The chosen approach enhances educational effectiveness through superior graphics capabilities, user engagement and interactive potential.

Host Tool Evaluation

	Low-Level APIs	**Scientific Libraries**	**Unity**
Graphics Capabilities	High	Low	High
Interactive UI Support	Low	Low	High
Performance Limit	Real-time	10 FPS	Real-time
Networking Support	No	High	High
Profiling Tools	Advanced	Basic	Advanced
GPU-Accelerated	Yes	Limited	Yes
Set-up Overhead	High	Low	Medium
Scripting Required	Yes	No	Yes (C#)
Complexity	High	Low	Medium
Learning Curve	Steep No prior experience	Gentle Prior experience	Moderate No prior experience

Table 3: Comparison of rendering frameworks for fluid dynamics simulation.

Unity was selected due to its real-time rendering capabilities, GPU acceleration, integrated networking, interactive UI-control, comprehensive support for shaders and post-processing effects.

Despite the initial learning curve caused by the team's limited experience with Unity and the greater overhead compared to lighter frameworks, its flexibility, performance, and extensive feature set ultimately justified the choice.

4.8.2 Data-Transfer Architecture

Host-based rendering means the data-transfer architecture is a critical component of this system. The latency and reliability of the data-transfer pipeline impact real-time simulation fidelity. An unoptimised transfer layer would undermine the benefits of offloaded rendering.

Client-Server Model

To support a low-latency, persistent TCP connection between the FPGA and Unity frontend, two TCP client-server configurations were evaluated.

In the first configuration, the FPGA operates as the TCP server and Unity as the TCP client. Unity initiates and manages network connection at runtime, aligning well with its event-driven lifecycle. Offloading connection initiation to Unity keeps the FPGA passive with respect to session handling, reducing FPGA-side resource requirements. This approach also allows the FPGA to support multiple simultaneous connections without additional logic, which is particularly beneficial for integration with the drawing application.

The second configuration designates the FPGA as the TCP client and Unity the TCP server. This requires the FPGA to wait for the Unity server to be initialised and listening before attempting to connect. The FPGA must implement more complex client-side logic which integrates less naturally with the Jupyter notebook workflow than the client-side Unity approach. Critically this configuration does not natively support multiple device connections without additional FPGA complexity.

Consequently, first model was chosen to provide a flexible and streamlined TCP network stack.

Push-Based Frame Delivery

A low-latency frame delivery model was required to fulfil the real-time performance requirements of the project brief.

A push-based transmission model was designed, where the FPGA transmit frames immediately post-computation. This maximizes throughput by eliminating idle time waiting for client requests, thereby reducing back-pressure and buffering requirements on the FPGA.

The pull-based alternative was dismissed due to the latency introduced by the round-trip overhead of additional Unity client requesting.

Binary Packet Format

TCP's built-in stream ordering and error checking eliminated the need for additional transport or application-level headers, allowing only the payload to be transmitted. This minimized transmission overhead and reduced parsing costs.

Each frame comprises sequential cell data in row-major order:

8-byte Packet			
ρ	$\lvert u\rvert^2$	u_x	u_y
2 bytes	2 bytes	2 bytes	2 bytes

Table 4: Structure of an 8-byte data packet used in the simulation.

Unity-Side Handling

To ensure stable, real-time visualisation, the Unity-side design decouples data reception from the rendering pipeline, isolating network variability from user-facing frame updates. Employing asynchronous frame reception, allows the main thread to remain fully dedicated to rendering and UI. This prevents network delays from blocking the rendering pipeline.

A lock-free buffering strategy safely handles incoming bytes with minimal latency, enabling concurrent frame reads and writes without expensive locks. Using a fixed-size circular buffer prevents unbounded memory usage and reduces garbage collection overhead, ensuring the renderer consistently accesses the latest frame.

Additionally, the TCP client handles parsing of raw frame data, offloading low-level decoding from the Unity renderer, for fast, stable, and predictable real-time playback.

4.8.3 Visualisation Data Types

To fulfill the project's educational objectives, informative and engaging data types were selected for visualisation. The data types originate from two primary sources: raw output from the LBM solver and processed data derived from these outputs. This combination provides a more comprehensive learning experience for users.

Density Field ρ

- The density field is visualised using a colour map. The fluid density represents the distribution of fluid mass highlighting regions of accumulation (high density) and voids (low density).

Velocity field $|u|$

- The magnitude of the velocity field is also visualised using a colour map. Users can identify regions of rapid versus stagnant flow.
- Additionally, streamlines tracing the trajectories that particles would follow through the fluid are displayed. This helps users understand the flow structure, including flow paths, circulation patterns and emerging vortices. The mathematical implementation of streamlines can be seen in Figure 15.

$$\boxed{\frac{dx}{dt} = u_x(x,y) \quad \text{and} \quad \frac{dy}{dt} = u_y(x,y)}$$

Figure 15: Streamline ODEs for x and y positions.

Vorticity field ω

- The vorticity field is again visualised by a colour map, showing the distribution of local rotational motion within the fluid. Vorticity is an important property to understand phenomena such as eddies, turbulent mixing, and flow separation. The mathematical representation of the vorticity is shown in Figure 16.
- Complementary iso-contour lines highlight regions with specific vorticity magnitude. This provides a clear, quantitative reference identifying the strength and extent of the rotational feature within the flow.

$$\boxed{\omega(x,y) = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}.}$$

Figure 16: Kinematic definition of 2D scalar vorticity

4.8.4 Drawing Application

The drawing application was designed to provide an intuitive and versatile interface for users to create and customise barriers for fluid simulation experiments. Its primary design goal was to accommodate a wide range of user preferences and skill levels while ensuring that the output remained compatible with the simulator's technical constraints.

The interface is based on Python and leverages the Pygame library due to its robust support for real-time mouse interaction, and efficient rendering of primitives. Users are provided with multiple drawing tools, including freehand drawing, circle, polygon, and rectangle generators. This allows both precise geometric shapes and freeform barriers to be easily created.

To support rich user interaction, an advanced dragging feature was included that allows objects to be repositioned on the canvas after placement when pressed. This accommodates user experimentation with the option to place barriers at various locations within the fluid simulation or introduce multiple barrier shapes simultaneously.

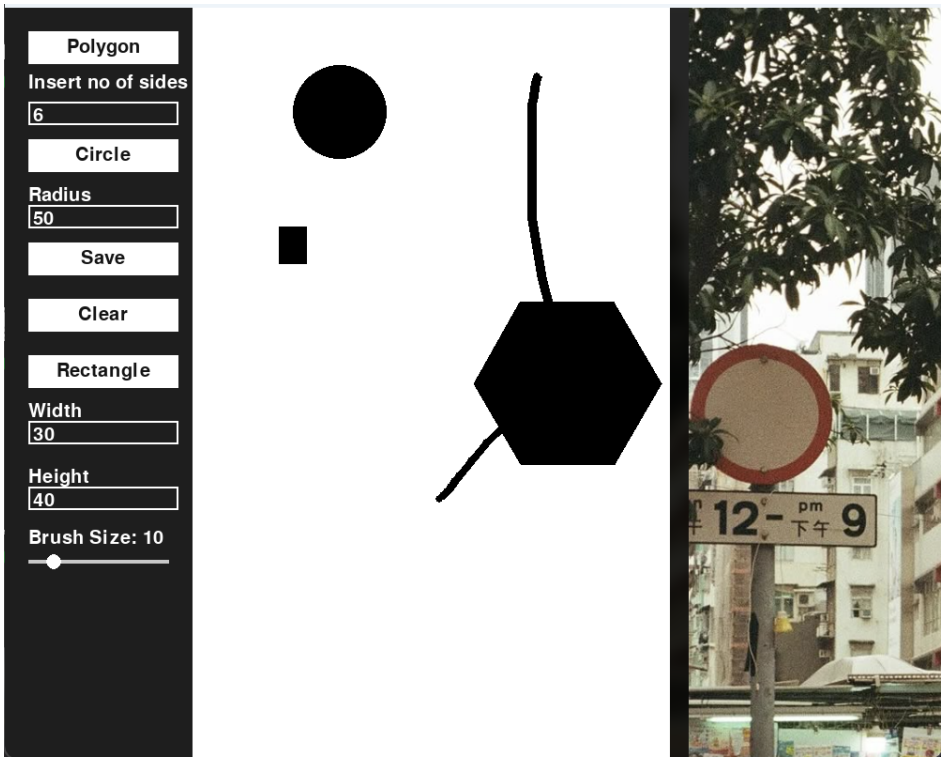

Figure 17: Drawing application demonstrating drawing features and image upload

The canvas was designed with a resolution of 800×800 pixels to ensure sufficient drawing detail for user interaction, while a downscaling process guarantees compatibility with the simulator's lattice grid size of 50×50. Black-and-white conversion and and packaging of the canvas output into a binary array (row major order) was also required for hardware-compatibility.

5 Development of subsystems

5.1 Solver Implementation

5.1.1 Single cell solver

The solver implements a large state machine show in Figure 20, with each of the states described below:

- **IDLE**: Checks if the number of generated frames is below the target step threshold and if so transitions to the STREAM state. To respect boundary conditions, the outer edges are excluded from streaming by adjusting the initial index and width count accordingly.
- **STREAM**: Checks if a barrier is present at the current index. If not, transition to STREAM_WAIT and start waiting for 2 cycles. If so, and the end of the frame is reached transition to BOUNDARY and reset the index, otherwise advance to the next valid index.
- **STREAM_WAIT**: After 2 cycles, the module computes the correct write addresses and enables for each direction, the writes the received memory data. At the end of the frame, it transitions to the BOUNDARY state and resets the index. Otherwise advance to the next valid index and transition to the STREAM state.
- **BOUNDARY**: Iterates though the second innermost edge of the lattice and writes the inward velocities of these cells to the boundary cells. Figure 18 shows cells visited during this stage and their order of iteration.

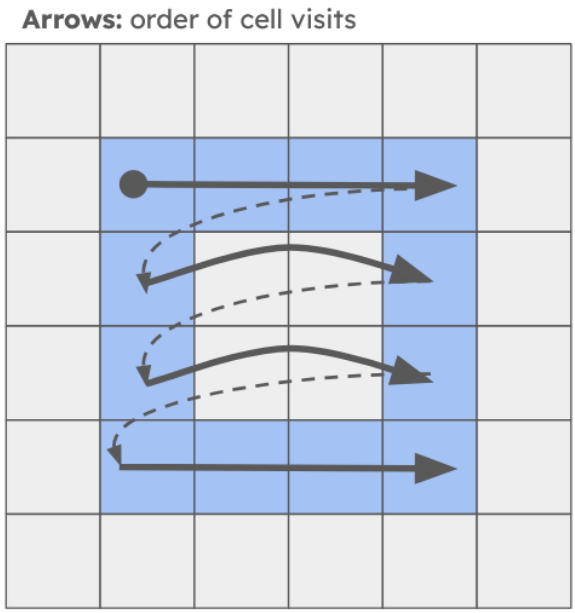

Figure 18: Cells visited by boundary stage

- **BOUNCE**: Checks if a barrier is present at the current index. If so, transition to BOUNCE_WAIT and start waiting for 2 cycles. If not and the end of the frame is reached transition to ZERO_BOUNCE and reset the index, otherwise advance to the next valid index. Barrier checks are limited to cells within 2 units of the boundary (Figure 19) to handle edge conditions. This is because if a barrier existed in the second-innermost cell, it would be bounced to the boundary, breaking the zero-gradient conditions.

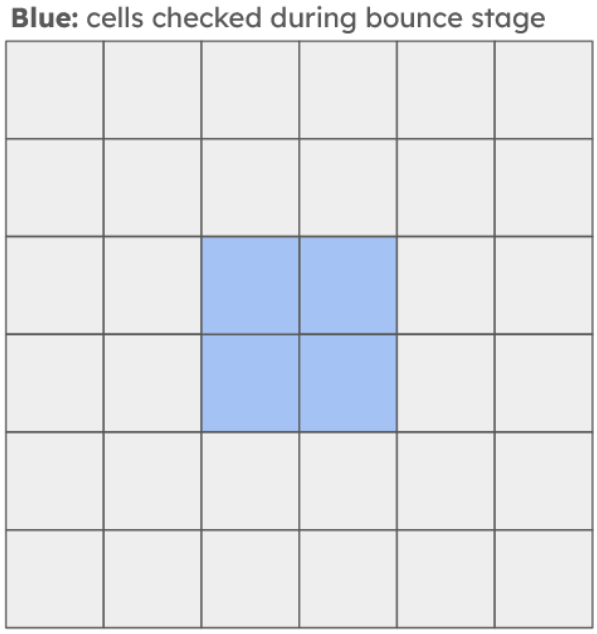

Figure 19: Cells visited in bounce stage

- **BOUNCE_WAIT**: After 2 cycles, computes the target write addresses and enables. Data read from each BRAM is written to the BRAM in the opposite direction (e.g. S data is written in the N BRAM). If the end of the frame is reached the ZERO_BOUNCE state is entered, otherwise the BOUNCE state is returned to.
- **ZERO_BOUNCE**: Checks if a barrier is present at the current index. If so, 0 is written to the cell. If the end of the frame is reached, the COLLIDE state is entered, otherwise advance to the next valid cell.
- **COLLIDE**: Checks if a barrier is present at the current index. If so, there is a two cycle stall before data is processed through the collider module, and memory is updated with the results. If the end of the frame is reached, the IDLE state is entered. Otherwise remain in COLLIDE and advance to the next valid index.
- **MEM_RESET**: This is the FSM's initial state, entered only when the reset is asserted low. It checks if a barrier is present in the current cell, if so the `init` input values are written into the cell. If the end of the frame is reached transition to IDLE otherwise remain in the COLLIDE state and advance to the next valid frame index.

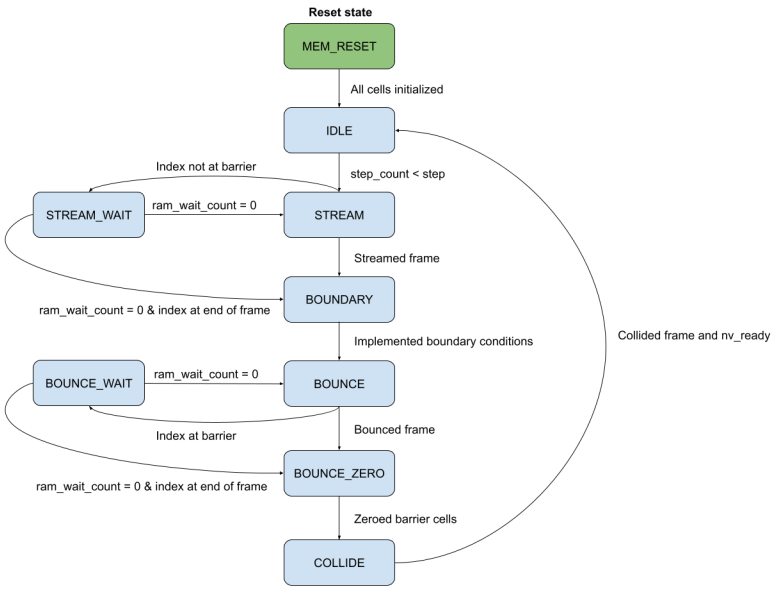

Figure 20: LBM State machine

5.1.2 Parallel Solver

The main difference from the single-cell solver is that this design reads from and writes to multiple cells simultaneously, implemented via two modules: the LBM controller and LBM parallel solver. The LBM controller manages indices for each parallel solver and BRAM write addresses, while also controlling write enables to determine which data is committed to memory.

Two new states, STREAM_READ and BOUNCE_READ have been introduced branching from the existing STREAM and BOUNCE states. These new states preload data from memory into registers to prevent unintended overwrites when individual solvers have differing write enables. Additionally, the existing state logic has been updated to index operations by the number of parallel solvers. In the WAIT states, if a write enable is high, new data is written to the target cell, if low, the previously buffered data is retained and written back.

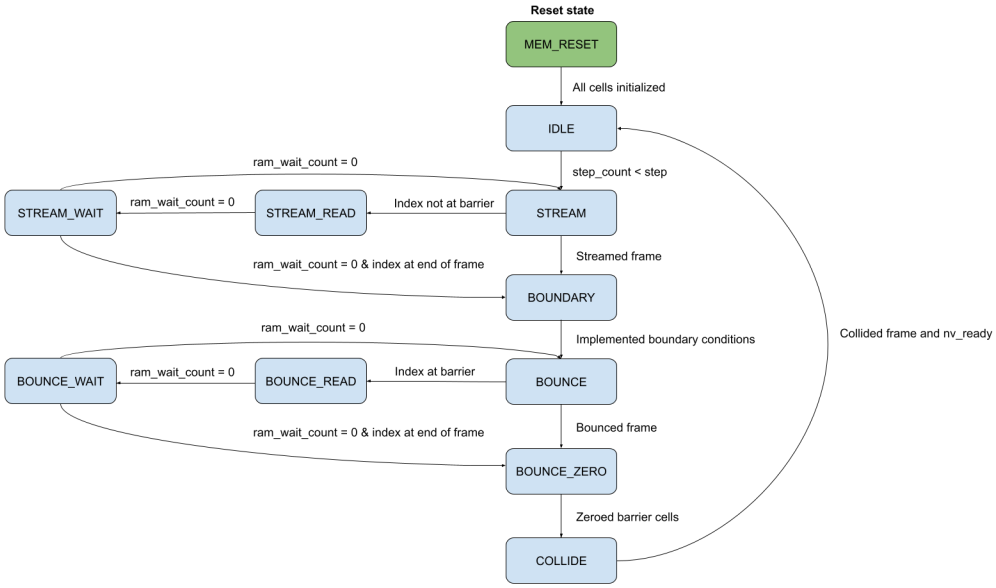

Figure 21: LBM Parallel State machine

5.1.3 Collider Implementation

Two versions of the collider were developed: a pipelined and a non-pipelined implementation. This dual design was chosen to ensure both accuracy and ease of debugging.

The first step involved implementing the multiplication algorithm, written as a Verilog function.

```
function automatic signed [15:0] multiply(input signed [15:0] a, b);
    reg signed [31:0] product;
    reg signed [15:0] shifted;
    begin
        product = (a * b) + round;
        shifted = product >>> 13;

        multiply = (product > 32'sh10000000) ? 16'sh7FFF :
                   (product < 32'shf0000000) ? 16'sh8000 :
                   shifted;
    end
endfunction
```

Figure 22: Multiply function

To gain a clearer understanding of the collision equations, a reference implementation was developed in Python (`collision.py`). This served as a useful tool for validating the Verilog implementation by comparing outputs. Values were converted to Q3.13 fixed-point format to match the hardware representation. Initial versions encountered issues such as overflow, which were traced to the absence of clipping during arithmetic operations.

In the pipelined version, multiplication operations that had no data dependencies were identified and scheduled to execute in parallel. This allowed for greater throughput. The non-pipelined collider could only be clocked at 10 MHz, whereas the pipelined collider operated at 100 MHz. The pipelined design consists of 21

stages, introducing a small setup delay. However, this delay is negligible in practice, as each 50×50 frame requires approximately 2500 × 3 cycles (accounting for RAM read delays), making the one-time pipeline fill latency insignificant relative to the overall computation time.

The Vivado Timing Analysis Report was used to verify that the pipelined collider could reliably operate at 100 MHz. From the timing diagrams, it was observed that the total combinational delay was reduced significantly—from approximately 82 ns in the non-pipelined design to around 10 ns in the pipelined version.

While the pipelined collider consists of 21 stages, which introduces a latency of 210 ns from input to output, this delay is mitigated in the broader context of the Lattice Boltzmann Solver (LBMSolver). The overall number of clock cycles required to compute a frame is also affected by other modules such as stream, and bounce. Through experimentation, it was found that the higher clock frequency enabled by pipelining led to a net reduction in the total time required to process each frame. This confirms the pipelined collider's effectiveness in improving overall simulation performance.

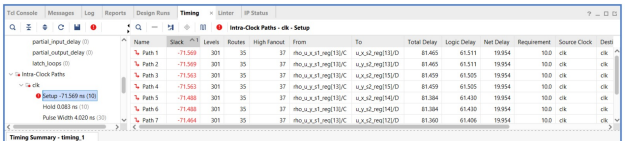

Figure 23: Non-Pipelined Collider Timing Analysis

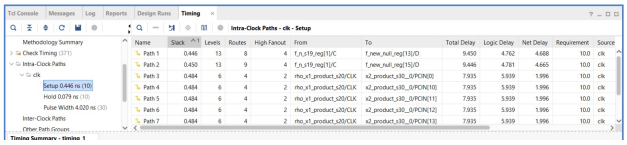

Figure 24: Pipelined Timing Analysis

```
// ------------------------
// Stage 16:
// ------------------------
`DECLARE(s16)

reg signed [31:0] f_eq_n_product_s16, f_eq_s_product_s16, f_eq_e_product_s16, f_eq_w_product_s16;
reg signed [31:0] f_eq_ne_product_s16, f_eq_sw_product_s16, f_eq_nw_product_s16, f_eq_se_product_s16;

always @(posedge clk or negedge rst) begin
    if (!rst) begin
        `RESET(s16)
    end else begin
        `PIPE(s15, s16)

        f_eq_n_product_s16  <= (rho_s15 * f_eq_n_intermediate_s15)  + round;
        f_eq_s_product_s16  <= (rho_s15 * f_eq_s_intermediate_s15)  + round;
        f_eq_e_product_s16  <= (rho_s15 * f_eq_e_intermediate_s15)  + round;
        f_eq_w_product_s16  <= (rho_s15 * f_eq_w_intermediate_s15)  + round;

        f_eq_ne_product_s16 <= (rho_s15 * f_eq_ne_intermediate_s15) + round;
        f_eq_sw_product_s16 <= (rho_s15 * f_eq_sw_intermediate_s15) + round;
        f_eq_nw_product_s16 <= (rho_s15 * f_eq_nw_intermediate_s15) + round;
        f_eq_se_product_s16 <= (rho_s15 * f_eq_se_intermediate_s15) + round;
    end
end
```

Figure 25: Example of Parallel Multiplication

5.2 FPGA Overlay Implementation

Prior to FPGA-based LBMSolver implementation, additional reasearch was undertaken to understand how to build a simple overlay in Vivado. The chosen learning tool was the implementation of a module capable of performing multiplication and addition, that was connected to the PS via GPIO. This time taken to understand how to interface with RTL streamlined later integration of the RTL onto the FPGA.

Initial PS-PL Interface The first FPGA-based implementation utilised 18 BRAMs connected to the solver, with all PS interfacing performed via GPIO subsubsection 4.7.1. This interface was query-based, so when the host wished to query frame data from the PL, it initialises a query session with the interface by raising the `HOST_TRANSMISSION` flag which prevents the interface from updating its buffers and allows the host to query different buffer addresses via AxiGPIO and retrieve the results via a separate AxiGPIO.

Furthermore, to ensure that values were taken from the collider at the right time, the interface also took in a `COLLIDER_READY` and `IN_COLLISION_STATE` input which was used to ensure the buffer was filled at the correct times. The FSM design of the interface is in Figure 26.

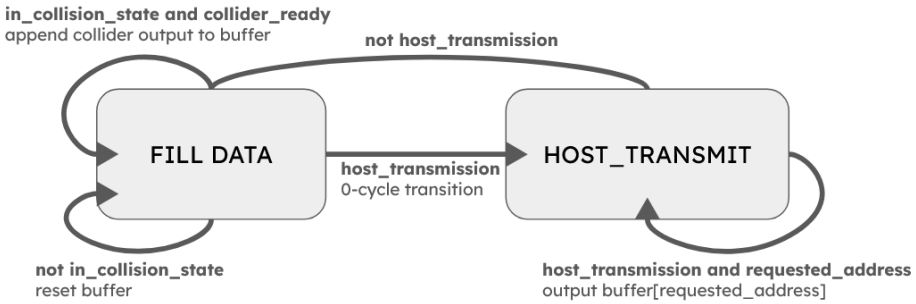

Figure 26: Initial PS-PL Interface

However, testing this interfacing method revealed the time to fetch one 50x50 frame was on average 0.191763162 seconds.

The second design iteration implemented a custom Axi-Stream Master which buffered data from the collider and sent its outputs via DMA to the PS. This allowed for faster data transfers. At a high-level, the module implements the ARM AXI Streaming protocal handshake. `TVALID` is asserted high whenever valid data are present, and `TLAST` signals the end of a transfer. Data is only transferred when both the `TREADY` input and `TVALID` signals are high.

Figure 42 shows the FSM that defines the design, which consists of two states `FILL_DATA` and `WAIT_READY`. Initially registers were used to buffer the Collider outputs, however the data-requirements for a 50 by 50 frame exceeded the available register slices. Instead the IP leveraged a BRAM block for buffering. This solution meets FPGA resource constraints and only adds a single cycle delay per `FILL_DATA` → `WAIT_READY` transition, which is negligible compared to the rest of the system.

To ensure that the AXI-Stream specifications were met, the system was designed like a Mealy-FSM, so output logic was combinational and depended on current state and current input. This allows for the correct behaviour of having the output change on the same cycle that `tready` is high.

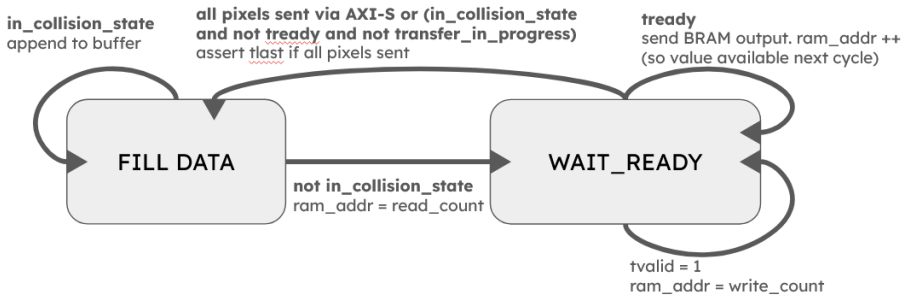

Figure 27: DMA Axi-Stream Master

This system took an average of 0.0010 seconds per 50x50 frame. A comparison between both interfaces in Figure 28 makes the improvement clear.

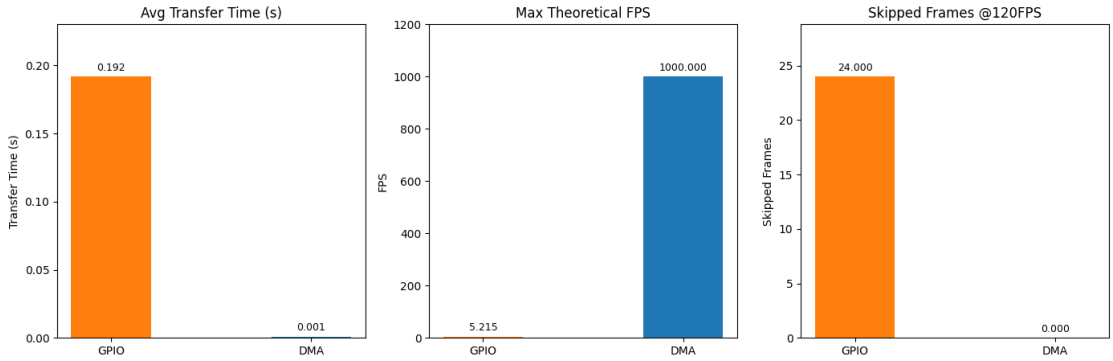

Figure 28: Transfer time, Maximum FPS, and number of frames needed to be skipped to render at 120FPS for each method

Due to careful early design of the PS-PL interfaces, the team encountered minimal issues integrating their Verilog RTL designs with the Pynq platform. Early integrated testing further mitigated risks, preventing a challenging transition from a large RTL codebase to a full Vivado project. Additionally, by excluding non-essential template overlay features (e.g., rendering, Arduino interfaces), bitstream generation and compilation times were significantly reduced, enabling faster design iterations on the FPGA.

5.3 Cache System

As outlined in Section 2.5, the cache system design is realised in hardware through two custom Verilog modules `BRAM_ctlr` and `DDR_pixel`

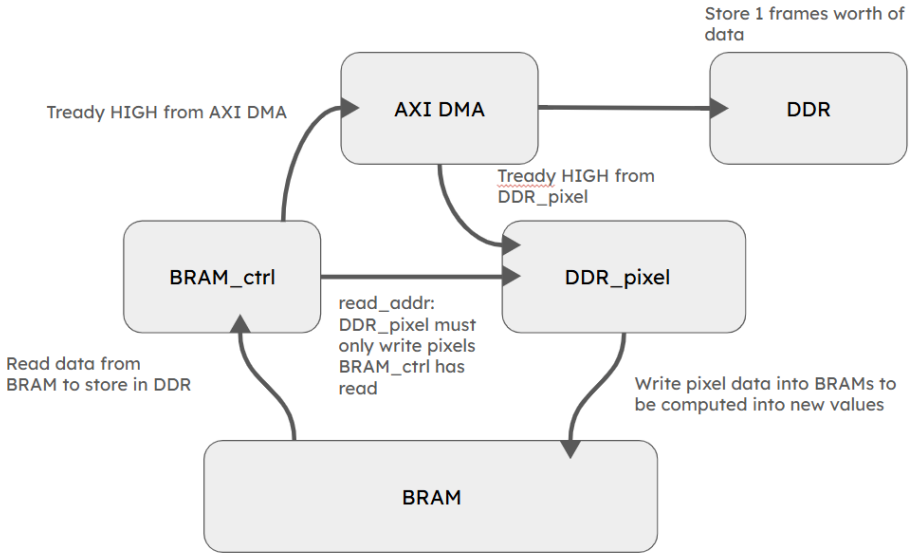

Figure 29: Cache system flow diagram

The BRAM controller operates as an AXI stream master implementing a two-stage FSM. The frame-ready signal is monitored and once asserted high, transition from an IDLE state to a streaming state READ_WAIT. In this active state, the directional values stored in each lattice cell from the 9 next BRAM arrays are sequentially read and packed into a single 144-bit AXI stream word. This stream word is output to the AXI DMA interface to be written to DDR.

The DDR writer is implemented as an AXI stream slave also implementing a two-stage FSM. The slave state is IDLE until incoming data is detected, at which point it transitions to the active state to accept the lattice cell data from the DMA stream. Each received packet is unpacked into the nine directional components and written to the corresponding BRAMs. The TREADY signal is asserted high when TVALID is high to indicate eradiness for the new input. The end of a transfer is detected via the AXI TLAST flag, resetting the module internal state for the next frame.

To avoid overwriting conflicts between the modules, the DDR writer only commits data to BRAM if the BRAM read address is higher than the BRAM controller write address.

The data exchange is managed from the Pynq Jupyter notebook interface, which configures the AXI DMA.

5.4 Unity Frontend

The Unity frontend architecture is structured into three components (GameObjects) handling communication, rendering, and user-interactions respectively.

5.4.1 TCP Client GameObject

Connection Initialisation

On startup, the client establishes a TCP network stream connection using a pre-configured IP address and port. Once connected, it continuously listens for incoming data on a dedicated asynchronous thread. Handling network I/O asynchronously, Unity avoids blocking the main thread which is essential to maintain responsive rendering and user input. This allows the main thread to tolerate variable network latency without impacting real-time performance.

Frame Buffering

TCP does not guarantee that an entire frame is delivered at one, to handle this an offset pointer accumulates bytes until a complete frame is received, ensuring data integrity.

Complete frame data is stored in a dedicated `SimulationFrame` structure that encapsulates the raw byte data, processed 1D row-major data array, and a volatile ready flag that indicates whether data is valid for rendering.

To handle high-frequency data, the client implements a circular buffer containing three `SimulationFrame` instances. The buffer provides a producer-consumer decoupling mechanism that guarantees the data reception task and renderer never read and write to the same frame slot simultaneously.

```
public class SimulationFrame
{
    public byte[] rawSimulationData;
    public volatile bool isFrameReady;
    public FluidCell[] parsedFluidGrid;
}

1 reference
private void InitializeFrameBuffer()
{
    simulationFrames = new SimulationFrame[bufferSize];

    for (int i = 0; i < bufferSize; i++)
    {
        simulationFrames[i] = new SimulationFrame
        {
            rawSimulationData = new byte[simulationFrameSize],
            parsedFluidGrid = new FluidCell[simulationWidth * simulationHeight],
            isFrameReady = false
        };
    }
}
```

Figure 30: SimfulationFrame Class and Buffer Initialisation

Data Processing

The `MemoryMarshal.Cast` method is employed to parse the byte array as a span of 16-bit integers. This method avoids additional allocations and enables fast numeric conversion.

The parsed data is then converted from Q3.13 fixed-point to floating point values by multiplying by 1/8192. Each data packet corresponds to a single simulation cell, which is processed into a custom `FluidCell` structure. Each `FluidCell` contains explicit fields for the ρ, ux, uy, and $|u|^2$.

```
1 reference
private void ProcessRawFrameData(byte[] newFrameData, FluidCell[] fluidGrid)
{
    Span<short> frameData = MemoryMarshal.Cast<byte, short>(newFrameData);

    int byteIdx = 0;

    for (int i = 0; i < simulationWidth * simulationHeight; i++)
    {
        short rawRho = frameData[byteIdx++];
        short rawvelocitySquared = frameData[byteIdx++];
        short rawVelocityX = frameData[byteIdx++];
        short rawVelocityY = frameData[byteIdx++];

        fluidGrid[i].density = rawRho * fixedPointScale;
        fluidGrid[i].velocitySquared = rawvelocitySquared * fixedPointScale;
        fluidGrid[i].velocity.x = rawVelocityX * fixedPointScale;
        fluidGrid[i].velocity.y = rawVelocityY * fixedPointScale;
    }
    count += 1;
}
```

Figure 31: Data Parsing and Processing Loop

Concurrency Handling

Concurrency between asynchronous data reception and the main Unity thread is managed using volatile state flags, eliminating the need for traditional locks which introduce thread blocking and degrade real-time performance.

Synchronisation relies on these volatiles flags to ensure data is read and written without race conditions.

5.4.2 Rendering Manager GameObject

The rendering manager `GameObject` transforms simulation data into real-time visual outputs. It integrates multiple rendering techniques, custom compute shaders, and procedural drawing methods to perform all rendering on the GPU for maximum performance.

To provide full control over how the simulation is visualised and interactively adjusted, all shaders are custom-built. Implementing these shaders from scratch required advanced HLSL and compute shader development skills.

Initialisation

During the `Awake` phase, the script initialises all rendering components.

- A dynamically scaling quad mesh is generated as the base geometry, ensuring a one-to-one mapping between fluid grid cells and rendered texels.
- Streamline spawn positions are systematically computed to populate a uniform grid. To prevent visual synchronisation, each streamline is assigned a random initial age, distributing respawns over time.
- Physical bounds are defined including the maximum Mach number and valid ranges for density, velocity, and vorticity. These bounds set the default normalisation parameters for accurate visualisation.
- Initialises four render textures (`velocityTexture`, `densityTexture`, `vorticityTexture`, and `velocityMagnitudeTextu` with optimised formats and filtering settings for high-quality rendering.
- All compute shader kernels for fluid, streamline, and vorticity updates are located, and their thread group sizes, and dispatch dimensions computed.
- Shader properties are configured and bound to ensure the rendered output correctly visualises the fluid data in the selected mode.

Rendering Process

During Unity's `Update` cycles, the renderer retrieves the latest frame from the TCP client buffer. Fluid cell data is then loaded into GPU-accessible structures, and shaders are configured dynamically in response to user-interactions.

Compute Shaders

Compute shaders enable highly parallel GPU-based data processing, which is essential for real-time performance. Each shader partitions work into optimal thread groups for its specific computation target.

- **Fluid Compute Shader**

- Processes fluid cell data into velocity and density textures.
- Computes the velocity magnitude from the squared velocity.

- **Streamline Compute Shader**
  - Uses second-order Runge-Kutta (RK2) integration to advect particles and update streamline positions. RK2 estimates an intermediary velocity at the midpoint of a timestep, then computes the final position, capturing accurate curved flow paths at low computational cost.
  - Implements domain wrapping and samples local density values to validate streamline segments, maintaining continuity.
  - Uses a circular buffer to manage streamline data, enabling respawn cycles without per-frame GPU memory reallocations.
  - Maintains multiple compute buffers for streamline head positions, tail positions, head indices, ages, tail segment validity, and indirect draw arguments. This system manages the streamline lifecycle and rendering.

```
// -- Current state sampling --
float2 currentPosition = _StreamlineParticlePositions[particleIdx];
float2 currentVelocity = _VelocityTexture.SampleLevel(sampler_linear_repeat, currentPosition * _InverseTextureDimensions, 0).xy;
if (!ValidateFluidDensity(currentPosition)) return;

// -- Midpoint state sampling --
float2 midpointPosition = currentPosition + 0.5 * currentVelocity * _StreamlineTimestep;
float2 midpointVelocity = _VelocityTexture.SampleLevel(sampler_linear_repeat, midpointPosition * _InverseTextureDimensions, 0).xy;
if (!ValidateFluidDensity(currentPosition)) return;

// -- Endpoint state sampling --
float2 endpointPosition = currentPosition + midpointVelocity * _StreamlineTimestep;
endpointPosition = (endpointPosition < 0) ? endpointPosition + _TextureDimensions : endpointPosition;
endpointPosition = (endpointPosition >= _TextureDimensions) ? endpointPosition - _TextureDimensions : endpointPosition;
if (!ValidateFluidDensity(currentPosition)) return;
```

Figure 32: RK2 Compute Shader Implementation

- **Vorticity Compute Shader**
  - Computes fluid vorticity by evaluating partial derivatives of the velocity field around each cell using finite difference approximations.
  - Outputs the computed vorticity into its dedicated texture.

```
float2 velocityWest = _VelocityTexture[clamp(texelCoord + int2(-1, 0), int2(0, 0), int2(_TextureDimensions.x - 1, _TextureDimensions.y - 1))];
float2 velocityEast = _VelocityTexture[clamp(texelCoord + int2(1, 0), int2(0, 0), int2(_TextureDimensions.x - 1, _TextureDimensions.y - 1))];
float2 velocityNorth = _VelocityTexture[clamp(texelCoord + int2(0, -1), int2(0, 0), int2(_TextureDimensions.x - 1, _TextureDimensions.y - 1))];
float2 velocitySouth = _VelocityTexture[clamp(texelCoord + int2(0, 1), int2(0, 0), int2(_TextureDimensions.x - 1, _TextureDimensions.y - 1))];

float dVy_dx = (velocityEast.y - velocityWest.y);
float dVx_dy = (velocitySouth.x - velocityNorth.x);
float vorticity = dVy_dx - dVx_dy;
```

Figure 33: Voritcity Compute Shader Implementation

Materials and Material Shaders

Materials serve as the interface between compute shader outputs and Unity's rendering pipeline.

- **Streamline Material Shader**
  - Manages visual properties such as hiding streamline segments during respawn and wrapping, and applies fading effects to enhance visuals.
  - Uses procedural indirect drawing to render streamline segments efficiently.

- **Fluid Material Shader**
  - Supports visualisation modes for density, velocity, or vorticity, using dynamic shader compilation via `#pragma multi_compile`.
  - Applies adjustable normalisation.
  - Implements vorticity contouring by computing partial derivatives of the vorticity magnitude and applying contour lines within a thresholded distance from the target vorticity level.

```
float vorticityMagnitude = abs(vorticity);

float vorticityDx = ddx(vorticityMagnitude);
float vorticityDy = ddy(vorticityMagnitude);

float vorticityGradientMagnitude = sqrt(vorticityDx * vorticityDx + vorticityDy * vorticityDy);

float distance = (vorticityMagnitude - _VorticityContourLevel) / (vorticityGradientMagnitude + 1e-5);

float contourMask = step(abs(distance), _VorticityContourHalfWidth);
vorticityColour.rgb = lerp(vorticityColour.rgb, _VorticityContourColor.rgb, contourMask * _VorticityContourColor.a);
```

Figure 34: Voritcity Iso-Contour Implementation

Textures

Dedicated colour textures translate normalised fluid data into colour gradients directly on the GPU, enhancing rendering efficiency and maintaining visual quality under high computational load.

5.4.3 User Interface

The UI controller facilitates real-time adjustment of fluid visualisation.

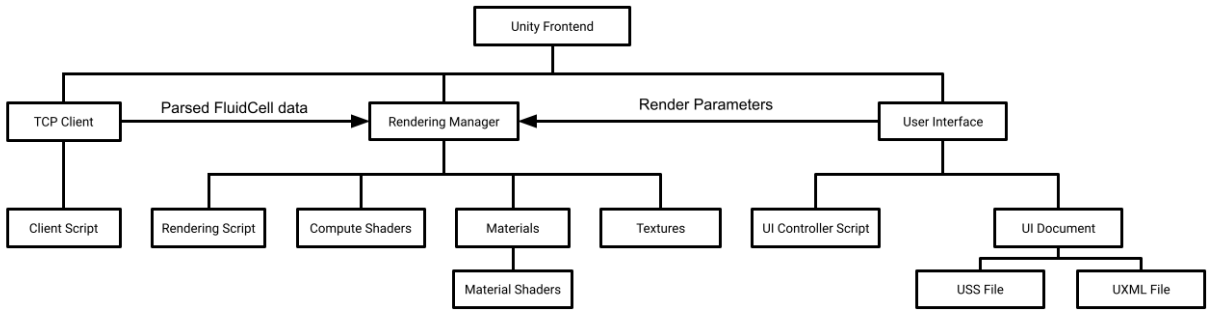

Figure 35: Frontend Architecture

UXML File

The UI is defined using Unity's XML-based UI systems.

1. The control panel provides visualisation configuration settings
   - Control panel button: Opens or closes the control panel to save space.
   - Toggles: Enable or disable colourmap rendering for Density, Velocity, and Vorticity.

- Sliders: Adjust normalisation parameters for each mode. An additional slider controls the vorticity iso-contour level.
- Streamline Configuration:
  - Radio buttons: Select streamline length.
  - Integer input field: Define the number of streamlines.
- Reset button: Instantly restores all settings to their default values.

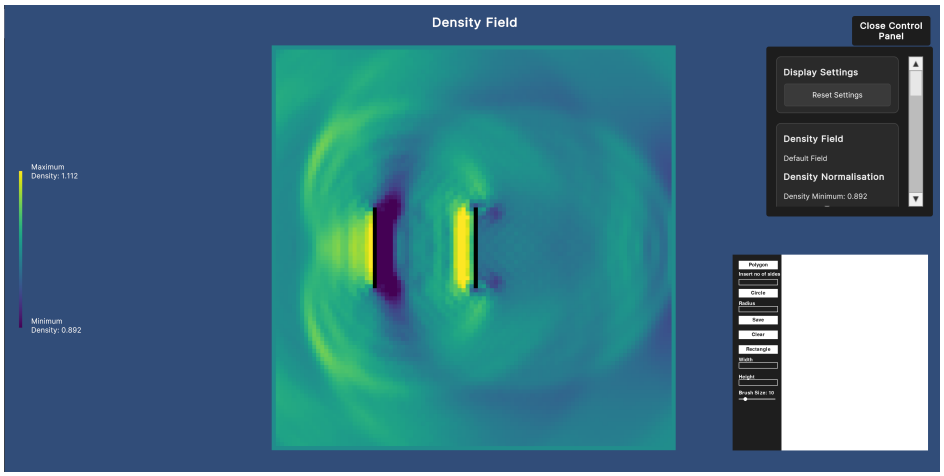

Figure 36: Frontend Styling with Scrollable Control Panel Open

2. Display elements enhance user comprehension with additional information.
   - Clearly labelled render fields and detailed legends.
   - A dynamic hover-over label displaying real-time simulation data.

USS File

Styling is handled via Unity's USS to create an intuitive, and space-efficient UI.

A high-contrast theme with a dark background, light text, and appropriate font size choice enhances readability. Control panel sections group related controls, while buttons and interactive elements follow coordinated styling for a cohesive appearance. Render legends and labels are positioned for visibility without obstructing the simulation. Non-essential UI elements remain hidden by default and are displayed only when needed to keep the visualisation focused on the render.

Controller script

The controller script handles all UI interactions and synchronises changes with the rendering pipeline.

- Mode Toggles
  - Density is the default render state. The controller listens for toggle state changes to enable or disable Velocity or Vorticity. When a mode is toggled, the controller instructs the renderer to switch shader keywords.
- Normalisation Sliders

– Sliders for each colourmap adjust the scale at which these fields are visualised. The controller captures slider value changes and immediately reconfigures the `fluid material shader`. The corresponding legend element is also updated to reflect the new scaling.

– An additional slider adjusts the value of the vorticity iso-contour. As with the normalisation sliders, the controller reconfigures the `fluid material shader`.

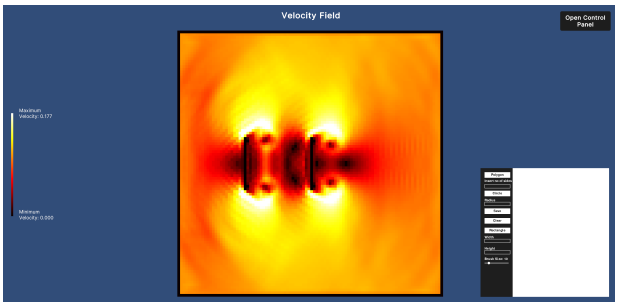

Figure 37: Velocity Field Render

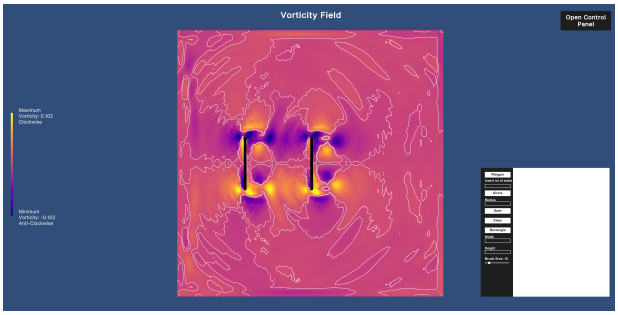

Figure 38: Vorticity Field Render with Iso-Contour

- Streamline Toggle
  - Enabling the toggle triggers the controller to request the renderer to initialise the streamline compute shader kernel, spawn positions, and associated GPU buffers.
- Streamline Configuration
  - Length selection and streamline count are managed by the controller, which reconfigures the material and compute shader parameters and instructs the renderer to release and reinitialise streamline resources.

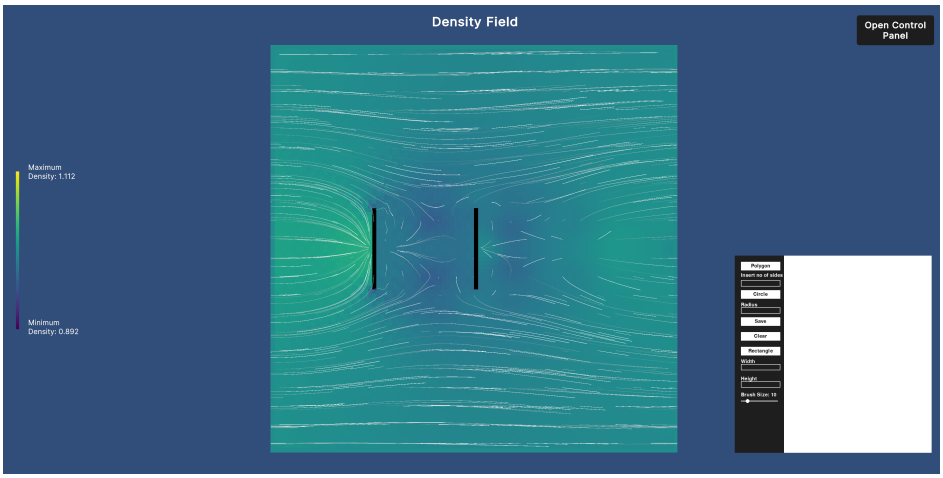

Figure 39: Density Render Including Streamlines

- Hover-Over Feature
  - A hover-over feature retrieves and displays fluid properties using a geometry-based approach rather than a collider-based one, ensuring minimal latency and scalability for arbitrary quad configurations.
  - The implementation proceeds as follows:

1. A ray is cast from the camera through the mouse's screen-space position.
2. Ray-plane intersection is computed to determine the world-space position, which is mapped to local quad coordinates and normalised to UV space $(0, 1)$.
3. The UV coordinate maps to a texel index in the render texture, retrieving cached data for display.

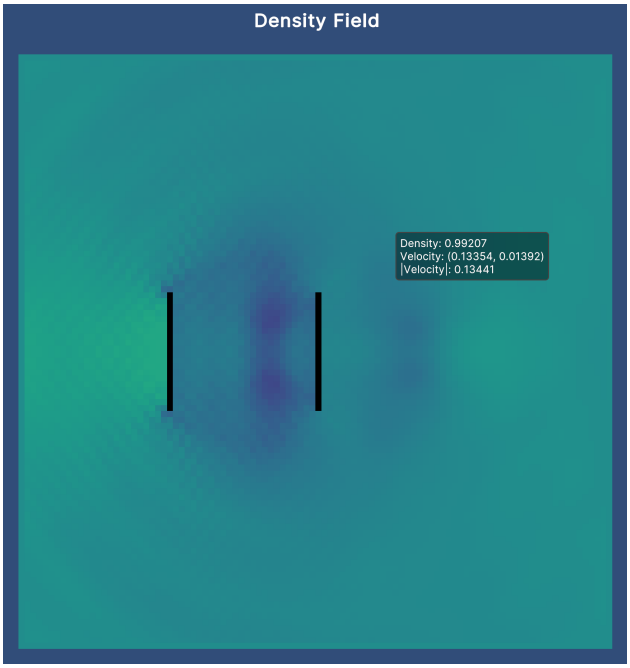

Figure 40: Hover-Over Information

- Reset Button
  - A single reset button reverts all visualisations to their default values.

5.5 Drawing Application

The initial implementation employed a straightforward approach where circles were drawn at each mouse coordinate. However, this led to discontinuities at high drawing speeds.

To address this, the implementation was refined to interpolate between consecutive coordinates with lines, and small overlapping circles were rendered at line junctions to maintain smoothness even at sharp corners. This was an important update because the gaps in the lines would significantly alter fluid interaction with the barriers.

Similarly to the host rendering platform, data transmission utilised TCP sockets where the FPGA was designated the server and the drawing application the client. A pull-based model was implemented to ensure that the FPGA continuously listens for updates, which is critical as barrier changes are asynchronous and must be handled in real time.

A custom IP, developed in Vivado manages the incoming data. Due to the AxiGPIO's limitation of receiving only 32 bits at a time, the IP is designed to accept 25 bits per transfer and stores them sequentially in a 2500-bit register. The `axi_ready` flag controls write operations, ensuring data integrity by signalling when new packets are available. To maintain correct bit ordering, the TCP server assigns a sequential word address

to each packet. Furthermore, the IP verifies that the `in_collision_state` flag is low before applying the new barrier, preventing mid-collision updates that could destabilise the fluid simulation.

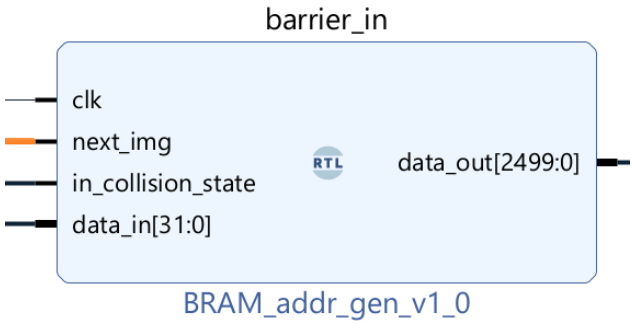

Figure 41: Barrier IP

5.6 Integration of drawing software and frontend

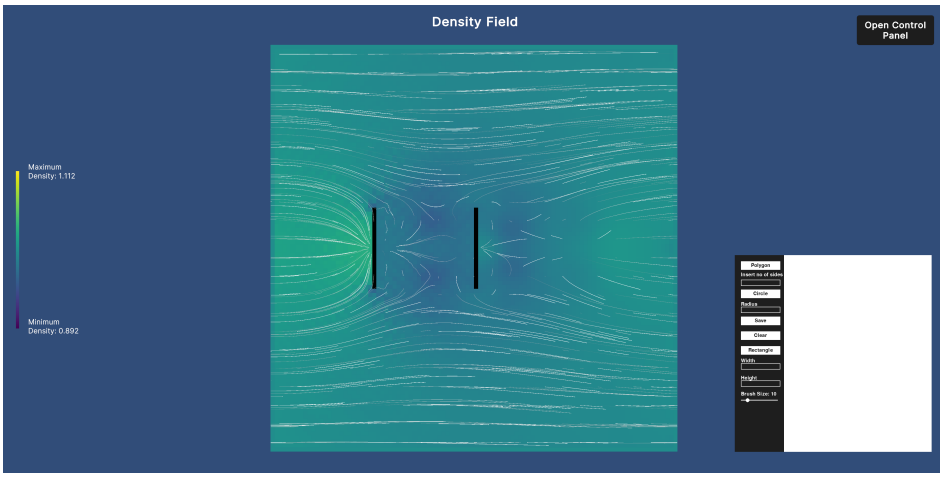

Figure 42: Integrated product

The integration was achieved using low-level Win32 API functions to establish interoperability between the independent rendering contexts.

Window Handle Acquisition

- The native window handle (`HWND`) of the Unity application was obtained using the `GetActiveWindow()` function from the Windows User32 API. Similarly, the Pygame window handle was accessed using `pygame.display.get_wm_info()`

Window Reparenting

- The Pygame window was created in borderless mode (`pygame.NOFRAME`) to eliminate the default window chrome. Using the (`SetParent`) function from the Win32 API, the Pygame window was then reparented to the Unity window, effectively making it a child component within the Unity graphical hierarchy.‘

Positioning and Layout Management

- The Pygame window was created in borderless mode (`pygame.NOFRAME`) to eliminate the default window chrome. Using the (`SetParent`) function from the Win32 API, the Pygame window was then reparented to the Unity window, effectively making it a child component within the Unity graphical hierarchy

Preservation of Interactive Functionality

- The Pygame event loop remained active within the embedded context, maintaining support for interactive features such as freehand drawing, polygon manipulation, image dragging, and binary canvas export. This ensured that the core functionality of the drawing tool was retained without modification.

The resulting system allows the Pygame application to operate natively within a Unity interface, enabling a hybrid toolchain that leverages Python-based 2D graphics capabilities alongside Unity's advanced 3D rendering and UI systems. This approach demonstrates a practical method for cross-platform integration between disparate graphical frameworks using OS-level window management techniques.

6 Performance Evaluation

6.1 Solver Evaluation

A prototype LBM was implemented in python which was crucial for testing the solver. The values from the Verliog implementation were tested against values output by the python LBM to identify simulation errors.

The first stage of testing provided the solver with equilibrium values, and a barrier along the outside edge of the simulation. To facilitate testing, a custom RAM module was created to emulate the FPGA BRAMs - for accuracy the RAM module also included a 1 cycle read delay. An additional state MEM RESET was added specifically for testing, to allowed control over the initial RAM values from the test bench. This was simulated using Verilator, and the waveform was inspected for deviations from equilibrium in GTKWave.

For the following test stage, the solver was implemented on the FPGA and given small eastward values. This setup was rendered in python using the prototype, then the outputs of the renders were compared to confirm both matched. This stage was utilised to identify solver mistakes, and prompted the implementation of boundary stage due to errors identified around the edge of the simulation. This stage resulted in a fully verified and correct solver.

The parallel solver was subject to the same testing process, initially with a single parallel LBM solver to identify solver errors, and then with the entire parallel system to identify integration errors.

Both the single-cycle and parallel solver meet the cycle requirements as less than 40 cycles per cell are required to compute a single frame. However, the resolution target could not be met with the limited BRAM resources, leading to the development of a caching system using the DDR RAM.

6.2 Collider Evaluation

The python prototype was extended to generate a reference video file, allowing visual validation of expected fluid behaviour. This reference was instrumental in understanding the correct dynamics of the LBM and served as a benchmark for verifying the hardware implementation.

During testing, it was determined that clipping was not required when computing the reciprocal of density ($1/\rho$). This insight allowed for a reduction of 6 pipeline stages, significantly simplifying the hardware and improving efficiency without compromising accuracy.

```
TEST_F(ColliderPipelinedTestbench, StrongNorthwardSpeed) {
    collider->f_null = 0x0E38;
    collider->f_n    = 0x05C0; // Boosted north
    collider->f_s    = 0x0180; // Weakened south
    collider->f_e    = 0x038E;
    collider->f_w    = 0x038E;
    collider->f_ne   = 0x0120;
    collider->f_se   = 0x0090;
    collider->f_sw   = 0x0090;
    collider->f_nw   = 0x0120;
    runSimulation(21);
    collider->eval();       You, 2 weeks ago • debug collider
    tfp->dump(ticks++);
    EXPECT_NEAR(collider->f_new_null, 3322, 2);
    EXPECT_NEAR(collider->f_new_n,    1417, 1);
    EXPECT_NEAR(collider->f_new_s,     671, 1);
    EXPECT_NEAR(collider->f_new_e,     831, 1);
    EXPECT_NEAR(collider->f_new_w,     831, 1);
    EXPECT_NEAR(collider->f_new_ne,    434, 1);
    EXPECT_NEAR(collider->f_new_se,    120, 1);
    EXPECT_NEAR(collider->f_new_sw,    120, 1);
    EXPECT_NEAR(collider->f_new_nw,    434, 1);
}
```

Figure 43: Example of test used for Collider

The expected values are generated by a Python script, and there is some approximation error because NumPy uses 64-bit floating-point representation. In comparison, the Q3.13 representation introduces truncation errors, but the difference in the calculations only varies by a few bits.

Despite removing the clipping from $1/\rho$, the test cases were still passed, and the fluid simulation generated remained unchanged when the hardware was executed, confirming the effectiveness of the optimization.

All tests were conducted using Verilator for simulation and GTKWave for waveform inspection and debugging. The final pipelined collider met all performance and accuracy requirements: it supports high clock frequencies, produces accurate fixed-point arithmetic, and yields correct evaluations of the LBM equations, contributing to a reliable and efficient fluid simulation.

6.3 Unity Frontend Testing

Frontend development employed physically accurate data from a Python-based LBM prototype before the FPGA implementation was ready. This ensured that visualisations were validated against reliable simulation results, enabling early detection and correction of rendering anomalies.

After establishing the core rendering pipeline, performance analysis enabled architectural optimisation. Initial code refactoring targeted the adoption of efficient data structures and high-performance computation techniques, laying the groundwork for scalable frontend rendering.

Unity's built-in profiler was employed to evaluate render frame rate and memory utilisation. The initial performance benchmarks were encouraging: the frontend consistently maintained the target rate of 60 FPS.

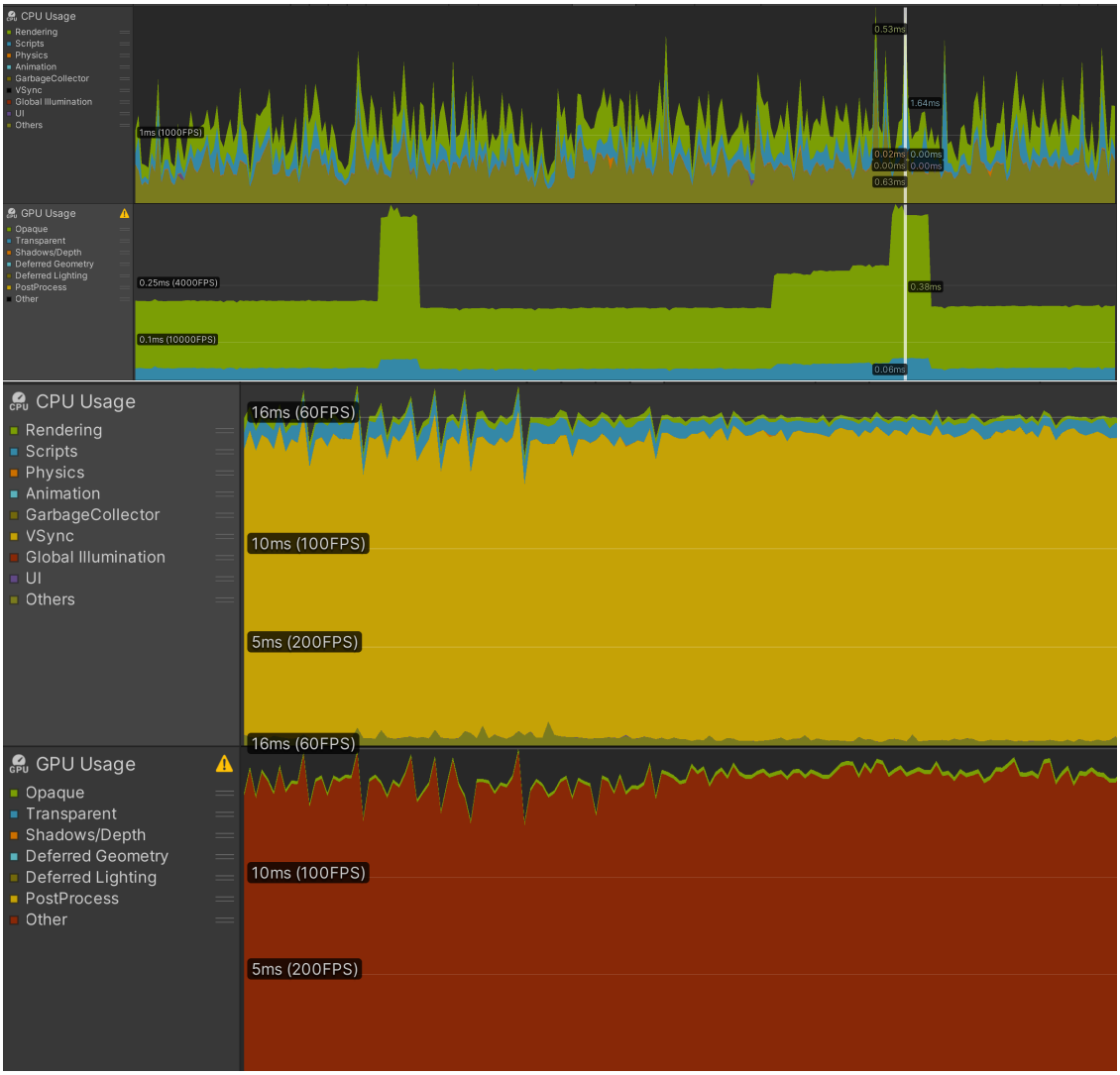

Figure 44: Unity profiler inital performance

However, memory profiling revealed excessive memory-usage spikes during runtime due to inefficient dynamic data handling and high garbage collection (GC) allocations. The final optimised implementation, described in Section 1.4, significantly reduced memory volatility during frame updates. Spike frequency decreases by factor $10 - 20$ and magnitude decreases from 58.6 KB to 2.4 KB. The earlier build also showed high graphics and driver memory usage due to inefficient management of multiple overlaid renders. This has subsequently been addressed, lowering graphics memory from 65.2 MB to 6.5 MB while preserving visual streaming quality. Together, total memory usage has been reduced by approximately 20% from 426.9 MB to 337.0 MB.

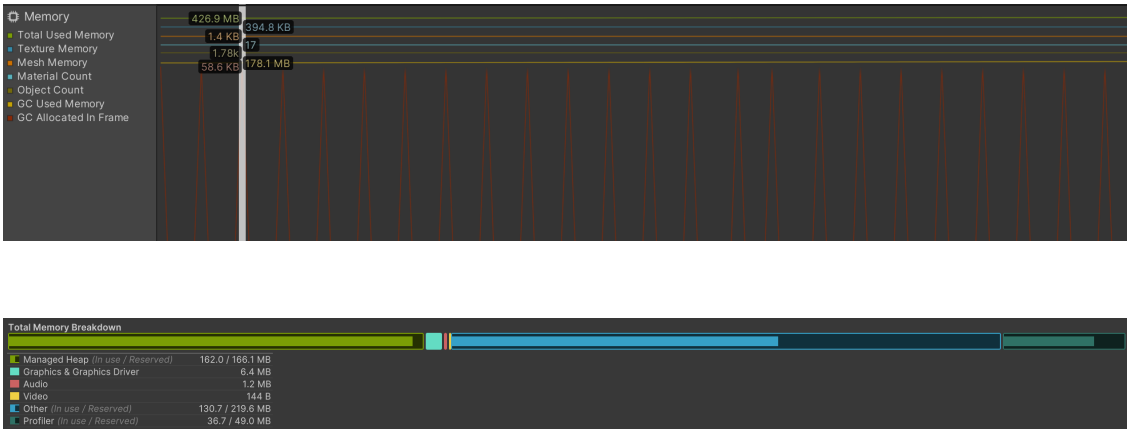

Figure 45: Memory profiler initial performance

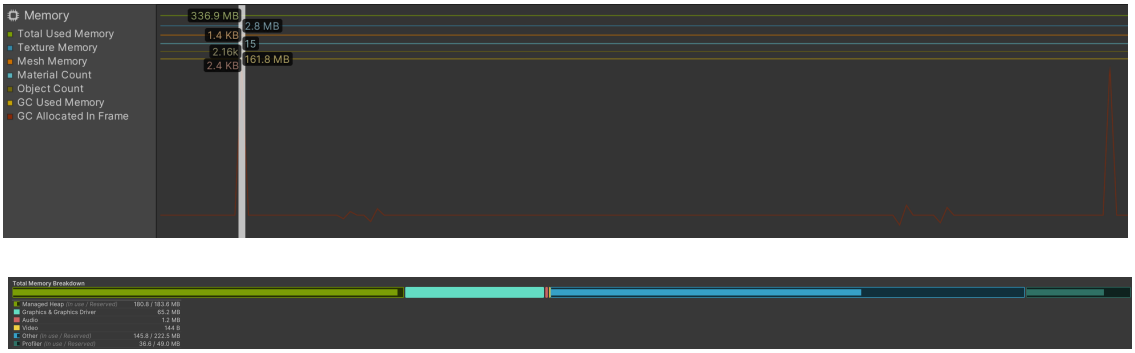

Figure 46: Memory profiler final performance

Unity Profiler performance testing concluded with the frontend achieving stable and responsive rendering at the screen-refresh rate of 60 FPS. The results demonstrate that the current architecture surpasses performance targets and, when synchronised with the screen refresh rate, operates as specified.

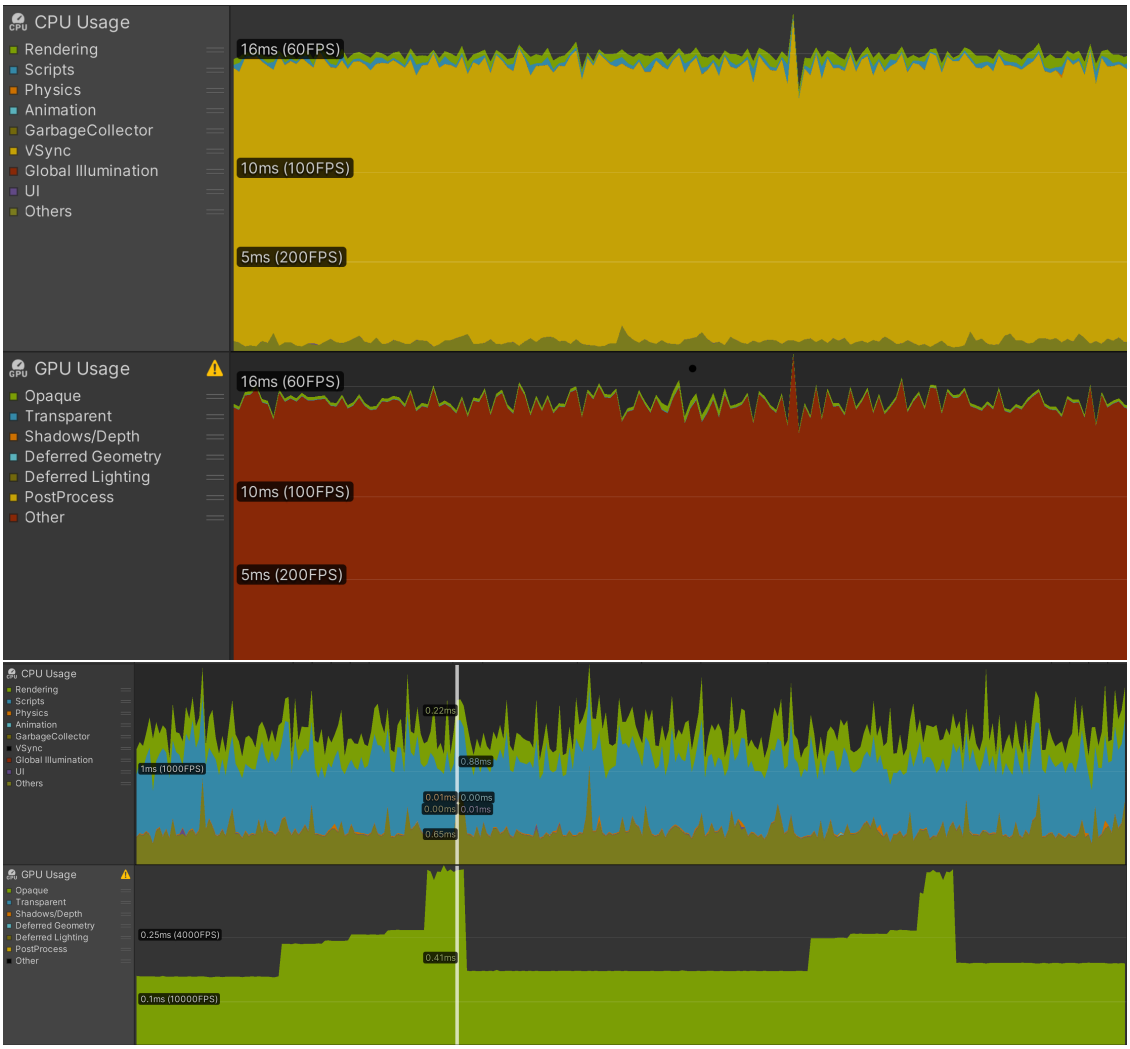

Figure 47: Unity profiler final performance

6.4 Drawing software Evaluation

During the design process, following an initial prototype, several people were invited to try and use the product, and a questionnaire was then provided.

The data below for the drawing software shows that although the majority of responders were able to effectively use the software, there were some limitations in terms of the easy of manipulation of shapes on the canvas, as well as a lack of clarity in terms of the drawing tools. This was mainly attributed to the fact that the shapes were initially centred solely at the centre of the canvas. Therefore, a dragging function was implemented to ensure that users would be able to freely move any shapes that they put onto the canvas. Further labels were also added to the interface showing the function of each button and sliders for clarity.

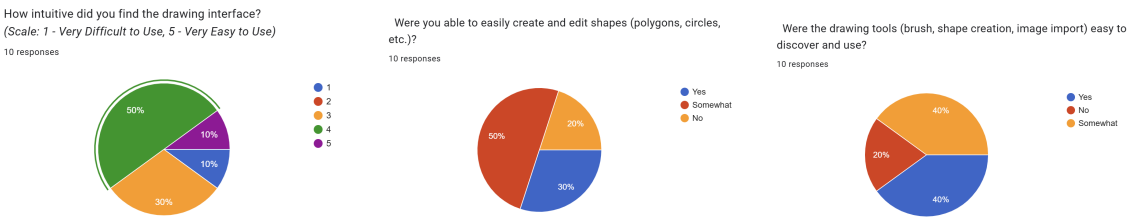

Figure 48: Drawing software questionnaire results

The majority of the responders were satisfied with the visualisation and responsiveness of the Unity frontend. Minimal changes were therefore made to the application.

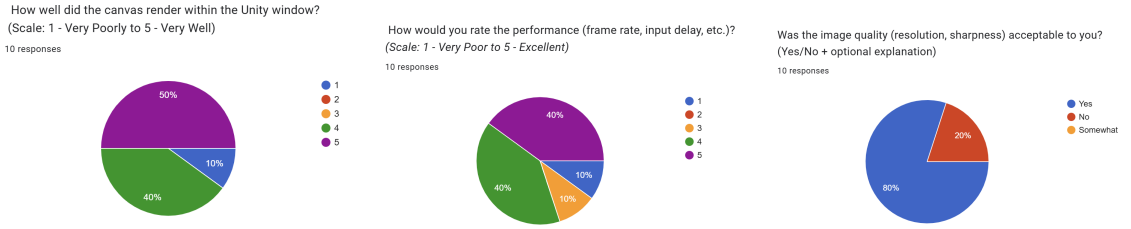

Figure 49: Unity frontend questionnaire results

6.5 Integrated product

The final product was evaluated in computational time to the version in Python, and the results are below. To allow for a fair test, the barrier was set as a 2x20 rectangle in the centre of the screen. Furthermore, we set the clock frequency to 50MHz, (below the maximum of 100Mhz), and two parallel solvers.

Python with Numpy	FPGA Rendering
0.0009845	0.0001329

Table 5: Seconds taken to compute one 50x50 frame

We see an almost 10x speedup in computational time. This number excludes the time required to display the data. In the Python case, this is the time taken to plot the matrix on Matplotlib, and in the FPGA case, this is the time taken to send the data to the renderer and for the renderer to render.

With more parallel solvers enabled and a higher clock frequency, we expect to see a much higher value, although we were unable to benchmark this due to time constraints.

Benchmarking the latency of the whole system with the time taken to display one frame, we see a further improvement of about 2.8 times. It is worth noting also that our frontend conveys a lot more information (velocity, density, streamlines) which require much more computation on the host end as compared to matplotlib, which only plots squared velocity, thus the speedup is much more significant.

Python with Numpy	FPGA Rendering
0.00871979	0.002238

Table 6: Seconds taken to compute and display one 50x50 frame

References

[1] V. Hunter Adams. *Lattice-Boltzmann in Python, C, and Verilog: An example of hardware acceleration via FPGA*. URL: https://vanhunteradams.com/DE1/Lattice_Boltzmann/Lattice_Boltzmann.html.

[2] Timm Krüger et al. *The Lattice Boltzmann Method: Principles and Practice*. Springer International Publishing, 2016.

[3] ARM Ltd. *AMBA ® AXI-Stream Protocol Specification*. ARM, 2010. URL: https://documentation-service.arm.com/static/60d5e2510320e92fa40b4788 (visited on 06/16/2025).