Non-Compressible Fluid Simulation Accelerator

Nick Sarkis (nas256), Christopher Graef (cdg79), Julia Currie (jbc262), Adam Macioszek (amm452)

1 Introduction

There exist many different physics simulators for a wide variety of applications. One such simulator is the Lattice-Boltzmann Model for computing the dynamics of a non-compressible fluid. As is the case for many physics simulators, the computations required for this simulation are not very complex; the simulation only consists of a constant number of multiplies and adds for each particle in the simulation to compute that particles next state value. Because of this simplicity, it is not too great of a task to create a software design which can perform these calculations. The problem, however, is the computation time. While the computation complexity is small, there is a very large quantity of computations. Each particle requires multiple adds and multiplies to be done in parallel for each iteration of the simulation. When the number of particles can easily scale up to be in the tens of thousands, the software simulation no longer becomes a viable option if you desire fast computation.

One of the main features of the Lattice-Boltzmann Model is that all of the computations can theoretically be done in parallel. There are no inter-particle dependencies in a single state; the next state is computed entirely using information from the previous state. This results in this sort of simulation being very attractive to optimize with a hardware accelerator. A design on the FPGA is capable of performing these simple calculations for each particle with a very high bandwidth. The individual computations will take more time than a CPU due to an FPGA's slower clock speed, but the possible parallel computing opportunity on the FPGA more than make up for this. We prove this by creating a design on the FPGA with a 13X speed up over a comparable software design, with the FPGA running at about 140MHz and with the CPU running at 4GHz.

2 Techniques

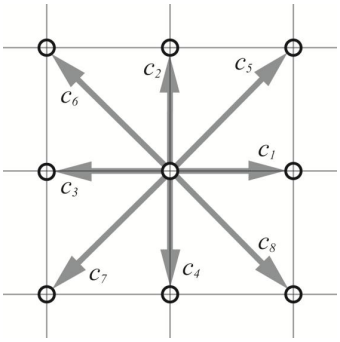
2.1 Lattice-Boltzmann Model

Lattice-Boltzmann is a physical model of fluid flow. It allows for the current state of the model to be calculated within one frame, as the density and velocity calculations for each pixel can all be calculated at once. Therefore, every particle can theoretically be computed in parallel. In Figure 2.1, each particle requires 9 parallelizable computations to determine both its pressure and the outflow into neighboring particles for the following frame.

3 Implementation

3.1 Python GUI and Visualizer

In order to efficiently create the first frame of the flow simulation we created a GUI in python. The program creates a 50 by 150 grid of buttons, as well as a reset and save buttons. Each button has an



$$f_1^{(eq)} = w_1 * [\rho + 3ux + 4.5uxx - 1.5uu] \quad (5)$$
$$f_2^{(eq)} = w_2 * [\rho - 3ux + 4.5uxx - 1.5uu] \quad (6)$$
$$f_3^{(eq)} = w_3 * [\rho - 3uy + 4.5uyy - 1.5uu] \quad (7)$$
$$f_4^{(eq)} = w_4 * [\rho + 3uy + 4.5uyy - 1.5uu] \quad (8)$$
$$f_5^{(eq)} = w_5 * [\rho - 3(ux + uy) + 4.5(uxx + uyy + uxy) - 1.5uu] \quad (9)$$
$$f_6^{(eq)} = w_6 * [\rho + 3(ux + uy) + 4.5(uxx + uyy + uxy) - 1.5uu] \quad (10)$$
$$f_7^{(eq)} = w_7 * [\rho + 3(ux - uy) + 4.5(uxx + uyy - uxy) - 1.5uu] \quad (11)$$
$$f_8^{(eq)} = w_8 * [\rho - 3(ux - uy) + 4.5(uxx + uyy - uxy) - 1.5uu] \quad (12)$$
$$f_0^{(eq)} = \rho - \sum_{i=1}^{8} \hat{f}_i \quad (13)$$

Figure 2.1: Lattice-Boltzman Model

number on it, which denotes what that pixel will be initialized to in the first frame. All of the buttons are initialized to a value of zero, or no disturbance. By clicking the button you can cycle the value to 1, indicating the presence of a barrier the flow cannot pass through. Subsequent clicks change the value from 2 to 5 in increments of 0.5, wrapping back to 0 after 5. Any value above 1 is represents a point of increased density. Once the user has created some barriers or disturbances they can press the save button, which will generate the csv file that represents the initial state of the fluid bed. If the user wishes to clear whatever changes he has made to the grid he can press the reset button and the buttons will be reset to 0.

In order to easily verify the correct behavior for each frame that we output from our simulation, we created a python visualizer. It take the large csv file that our simulation saves at the end and splits it into frames, which are then converted to individual images. We then scale up the image by a specified factor and assign color based of scale. The frames are then stringed together to create a GIF.

3.2 Baseline Software

Our first baseline software design performed all of the calculations for each new state in two stages. In the first stage, all of the outflows for each pixel were calculated. The outflows of each state in the Lattice-Boltzmann model only depend on the current velocity and density of the particle, so all of these calculations could be performed with the state information of just one particle. All of these outflows were then stored in an array.

With these values stored, blocking values could then be calculated. These calculations are performed by going to all particles which are considered blocks in the simulation. At each of these particles, the outflow in every direction was set equal to the inflow in the respective direction, and these values were again stored in the arrays holding the intermediate values.

The flow simulation could also be performed at the stage. The flow is calculated by first sending any pressure on either the left or right hand side of the simulation to the opposite side. Conceptually, this allows flow exiting the screen on say the right side to be reinserted on the left. Next, a constant was added to the pressure being inserted onto the left side particles and the same constant was removed from the pressure being reflected back from the right side particles. This inserts pressure on the left hand side of the simulation and an equivalent pressure is then removed on the right hand side. The resulting outflows are stored in the intermediate value arrays. The end effect of this step is a pressure gradient across the simulation, simulating a constant flow from left to right.

Finally, all of the intermediate outflows have been calculated, so the next state for each particle can be computed. By iterating through each particle, the next state of the particle can be computed using the past state of the particle as well as the values in the intermediate array. These next state particles are stored in their respective arrays, and this process can repeat to calculate the next frame of the simulation.

The above worked as a baseline, though we made alterations as the design developed. We determined that a more efficient solution on the FPGA would be a design which treated each particle state calculation as a convolution. In this case, all of the above calculations are still performed with the key

difference being when they are performed, In this model, all of the calculations were performed as required for inflows to a particle. This means there was only a single stage in the calculation, and no array was required to store intermediate outflow buffers. The actual number and type of calculations being performed were the same, though.

3.3 Baseline Hardware

Our hardware baseline was implemented as a hardware/software codesign using the Vivado HLS toolchain in order to synthesize part of our C-based design in hardware. We initially targeted the Zedboard for our design as we were familiar with the platform and therefore would allow us to quickly prototype various implementations in the early stages of our project.

Our initial design requires 4 individual steps: calculating outflow, simulate blocking (walls and barriers), simulate flow (insert particles at the top of tube, remove from the bottom), and finally calculating the resulting density and velocity vectors required to describe the new frame's state. These operations were pipelined using the HLS dataflow pragma, however little speedup was achieved due to the extremely high memory usage of the calculate outflow stage and the high computational latency of the frame state calculation stage.

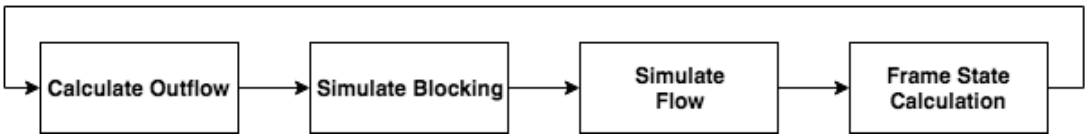


Figure 3.1: Hardware Baseline Flow

The hardware and software use the Xillybus FIFO-buffer framework to communicate with each other. The framework emulates two FIFO buffers between the FPGA and ARM core using the built-in AXI interface inside the Zync SoC. To kickoff the simulation, the host processor sends an initial state to the FPGA, consisting of densities and velocities for every point in the frame. For each frame onward, the FPGA sends the density at each pixel in the frame. It does not send the velocities as this data is irrelevant to the simulation's output and is only used as an input to the next frame. The FIFO allows the FPGA to stream values in one entry at a time, and was a major bottleneck for our design which had a fairly irregular access pattern. In order to be able to take better advantage of the parallelizable nature of this problem, the memory structures holding the arrays describing the system's state needed to be heavily partitioned. At this point, we realized that the overhead introduced by Xillybus was the main bottleneck in our design, as copying data to and from the FPGA alone was taking more than 75% of the overall execution time.

Given the hardware limitations of the zedboard and the speed limitations of the Xillybus framework, we decided that it would be best to switch to the SDSoC framework (for faster host-fpga communication) and to the ZC706 board (to allow more hardware-intensive optimizations).

3.4 Optimized Hardware

In order to increase data transfer speeds between the host processor and the FPGA, we utilized the SDSoC speedup to take advantage of the higher throughput that it can offer through the AXI bus. To better suit the SDSoC environment, we moved all our code intended to be implemented in hardware inside a function named *lb_accel_step(...)* (named due it performing a single simulation step using a lattice-boltzmann accelerator). This function takes three inputs: the previous state's density and velocity at each pixel, and produces three outputs: the new state's density and velocity at each pixel.

While switching to the SDSoC framework realized a 6x speedup compared to the Xillybus implementation, we soon pivoted to a design that enabled us to utilize a line and frame buffer combination, requiring only a single value from memory per cycle. In our initial design, we utilized an "outflow"

array of size $numPx_x * numPx_y * 8$ that stored the calculated outflow values during the "calculate outflow" stage. These values would then be used by the new state calculation. However, this introduces an unnecessary intermediate step for these outflow values, and this new design calculates them on the fly as they are needed.

Rather than storing the outflow values, we created a module that can calculate outflow in any direction. It requires the velocity and density of the pixel in question, and the direction of flow that is requested. This module also incorporates calculations for walls so blocking calculations are no longer a distinct step. Since blocking behavior is included in this module, the new state calculation stage implicitly includes it. The new state calculation is depicted below:

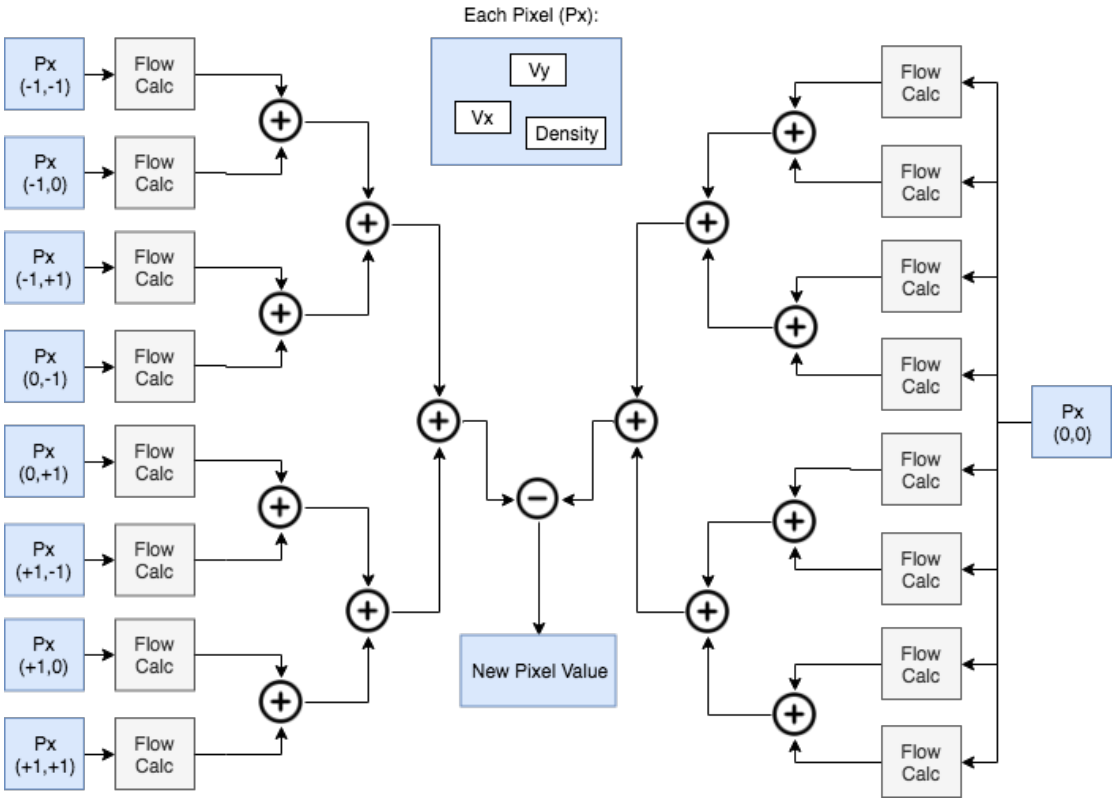


Figure 3.2: Calculating new state in Hardware

In this diagram, the velocity and density values of pixels in a 3x3 window around the target pixel are utilized to generate the velocity and density values of the pixel's next state. This calculation produces a single pixel's new values and each instance of this calculation is fully parallelizable over a single frame, since each *Px* in this diagram is from the previous frame.

The above diagram is essentially an implementation of a filter with some extra computation added in the middle. The flow calculation itself is multiplexes Any given pixel's next state is only dependent on the density and velocity values of all the pixels in a 3x3 window surrounding itself. This problem can therefore easily be modeled as a 3D convolution, with three window/line buffer pairs, for density and for each component of velocity. We decided to implement our own line buffers and frame buffers as our use-case only requires a very basic implementation of each.

The 3x3 window buffer approach works extremely well for the general case, since each pixel only needs information about its neighbors when calculating its next state. However, it does not work well for calculating flow, since pixels on the inflow side rely on pixels on the outflow side. This breaks the locality that was initially exploited in the window buffer-based design. To solve this problem, we perform the flow calculations in software while the hardware performs computation on pixels without any artificially introduce flow. After each call to a hardware function, SDSoC implicitly inserts a call

to a function that waits for the hardware accelerator to finish its task before continuing with the host program. This is to ensure correctness without any extra effort by the programmer. SCSoC provides the "async" pragma that allows the user to move this wait call. We use the following structure to perform flow calculation in software in parallel with the hardware accelerator.

```
#pragma SDS async(1)
lb_accel_step(...);   // Hardware Accelerated Function
calculate_flow(...);  // Do software flow calculations here
#pragma SDS wait(1)   // Move the sds_wait() call to here - wait for HW to finish
```

This works very well because there is no dependence between the software and hardware operations since they are both operating on the same frame. When the software finishes its part, it then waits for the hardware portion to complete. The software's job is computationally simple compared to the hardware's job, and generally completes much quicker than the hardware. The software flow calculation only occurs on the first and last rows, so if we make our simulation area much longer than it is wide, we can be sure that software will never bottleneck our design. We've found that a width:length ratio of 1:3 ensures this.

Overall, the interplay between hardware and software result in the following system:

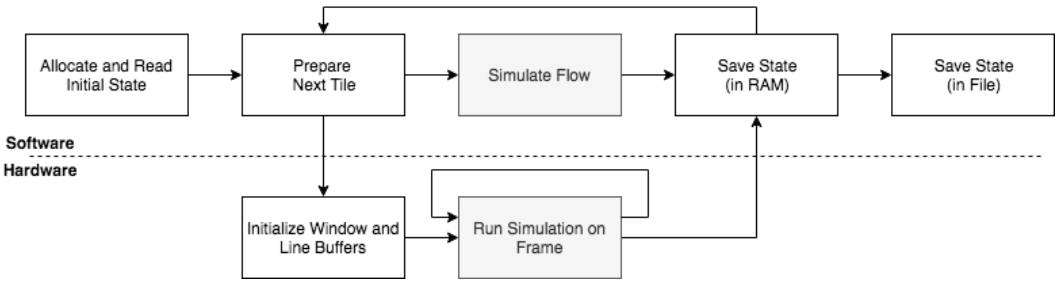


Figure 3.3: Optimized Hardware

4 Evaluation

4.1 Accuracy

The difference between simulations run with the floating point SW baseline and the fixed point hardware implementation was less than 1%. We implemented fixed point in hardware for performance reasons and because the accuracy was not a limiting factor. Additionally, we cast the resulting 32-bit number to a 16-bit number to decrease dump file size and speed up python execution. With this in place, there is no difference between the outputs produced by floating point and 32-bit fixed point designs.

4.2 Speed

To normalize speed metrics, we choose a sufficiently sized space that follow the width:height ratio of 1:3 whose benefits were described in Section 3.4. We simulate 100 pixels wide and 300 pixels tall for a total of 100 simulation steps, resulting in 3 million pixel operations total. We do not include file writing times in these numbers, as we wish to highlight to performance of the FPGA accelerator, not the attached drives.

We see a sizeable speedup on ecelinux compared to the ARM core due to its much larger power budget and more powerful floating point hardware. As will be shown in the following section, the convolutional ZC706 design uses less than the hardware available to it. We could therefore use the

Design	Execution Time (ms)	FPS	Speedup (Rel. Zedboard SW)
Zedboard Software (-O3)	17,023	5.87	1
Ecelinux Software (-O3)	1,334	74.97	12.12
Zedboard + Xillybus	?	?	?
ZC706 + SDSoC	1,415	70.67	12.04
ZC706 (conv) + SDSoC	110	909.09	154.87

Table 4.1: Execution and relative speedup of each design

`#pragma SDS resource(x)` pragma to tell SDSoC to make distinct copies of the accelerator. Our code could then use look something like the following:

```
for (int tile = 0; tiles < TOTAL_TILES; tile++) {
    #pragma SDS async(1)
    #pragma SDS resource(1)
    lb_accel( /* First half of tile */ );

    #pragma SDS async(2)
    #pragma SDS resource(2)
    lb_accel( /* Second half of tile */ );

    /* Flow calculation in software */

    #pragma SDS wait(1)
    #pragma SDS wait(2)
}
```

This would allow us to theoretically double our speed if AXI bus resources do not bottleneck the design too much. However, we experienced some issues with the SDSoC compiler while trying to use this feature, and were not able to successfully implement and test it. The speedups we were able to achieve were largely due to our ability to pipeline the majority of the design to an II=1, which reduced the overall load on the AXI bus and reuse hardware as efficiently as possible.

4.3 Hardware Usage

Design	LUT	FF	DSP	BRAM
Zedboard + Xillybus	7,954 (14%)	4,267 (4%)	87 (39%)	352 (125%)
ZC706 + SDSoC	81,926 (37%)	80,155 (18%)	696 (77%)	360 (33%)
ZC706 (conv) + SDSoC	28,994 (13%)	94,109 (21%)	253 (28%)	12 (1%)

Table 4.2: Resource usage for each FPGA design

The zedboard design was not synthesizable. Even without any optimizations, vivado_hls was unable to allocate enough space to hold the arrays required in our design for our target frame size. We decided not to pursue the work of software tiling the design since we focused the majority of our efforts on the SDSoC solutions.
The first SDSoC-based design required much more BRAM usage. This is because the accelerator stores 11 memory elements per pixel: density, x-velocity, y-velocity, and one for each of its 8 outflows in BRAM. The size of BRAM used grows with the number of total simulated pixels, and represents a major

limitation of this design. This design also uses significantly more DSP elements than the convolutional form. This is because we significantly unrolled loops that performed new state and outflow calculations, since we could not get a pipeline with a beneficial II due to memory limitations.
The last design modeled the problem as a convolution. Therefore, this design was able to take advantage of line/column buffers, which almost completely eliminated the BRAM usage on the device. The only memory elements on the device were the line and column buffers themselves, which fit nicely inside registers. Each cycle, a single value on each AXI input would be read and a single value would be written. The interface can easily support these speeds and therefore nearly eliminates all BRAM usage. Since the design is heavily pipelined, it makes more efficient use of the DSPs compared to the previous design, which unrolled loops in order to boost performance. As a result, the DSP usage drops significantly while performance increases. There is a small increase in flip flop usage due to the line and column buffer partitioning. It is also interesting to note that a single instance of this design could potentially fit on the Zedboard, if the pipelining was made slightly less aggressive to reduce DSP usage.

5 Project Management

Week of 10/30/18:

Decided upon fluid physics simulator as final project

Week of 11/6/18:

Created first software baseline of the simulator - Chris, Nick

Created python visualizer - Adam, Julia

Week of 11/13/18:

Created hardware baseline of the simulator - Nick, Julia

Developed blocking for first software baseline - Chris

Week of 11/20/18:

Second software simulator model developed (convolution) - Nick

Hardware moved to using SDSOC - Nick, Julia

Convolution model implemented in hardware, optimized with line/col buffers - Nick

Flow added to software model - Chris

Week of 11/27/18:

Implemented flow in software to work in parallel with FGPA state computation - Chris

Created GUI for creating initial conditions - Adam, Julia

6 Conclusion

We were able to successfully implement a hardware physics accelerator on an FPGA. The simulator we implemented incorporates a high degree of memory dependence between pixels, which bottlenecked our design for most of the project. However, we were able to redesign the project to utilize the line and column buffers presented in this class in order to almost completely remove any memory-related slowdown. After surmounting this obstacle, we were able to beat the performance of an Intel Xeon 5-2680 CPU with a base clock 2.50GHz by over 13x using a small ARM core and embedded FPGA (the ZC706) using significantly less than half of the available resources.

7 References

Nowicki, Claeson, SoC Architecture for Real-Time Interactive Painting based on Lattice-Boltzmann, (https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5724497tag=1)