Optimizing a Fluid Simulation

Jesse Kettmann, Charl-Pierre Marais

Introduction

This report covers our optimization and consequent speedup of a Lattice Boltzmann Fluid Simulation created by GitHub user “baderouaich”. Grid based simulations are the perfect example of applications the GPU does best. As such, when making said simulations, it is worth trying to offload as many calculations as possible onto the GPU. The base code of the simulation did not run on the GPU whatsoever, and since neither of us has any previous experience with GPU programming, we figured this project would be a great learning opportunity. Our goal was thus to optimize the simulation part of the project using OpenCL.

A peek at the code revealed that all of the operations on the fluid simulation were relatively simple, atomic and vectorizable, which in theory should make the code easy to port. Unfortunately the actual logic behind how the simulation works was very hard to grasp (even for baderouaich themselves, according to the comments). It is not strictly necessary to understand the functionality of the simulation to port it over, but there were a few situations in which we thought baderouaich had made a mistake, which we elaborate on in the discussion. In the end, we did manage to successfully port the simulation part of the code to the GPU and achieved a 2.57 times speedup. Note that this number can theoretically go much higher, as explained in the results section.

The structure of this report is as follows: First, we explain the base functionality of the simulation. We then dive into the methodology used during the optimization process. After that follow the results of our efforts, and finally, some discussion.

Base Functionality

Below we describe the base functionality of the project codebase, for as far as the project itself made it clear. We did not have time to research the Lattice Boltzmann method, nor is a deep understanding of its logic necessary for our purposes. We only describe the simulation side of the project (leaving out user input and rendering), as this is the only part we have optimized. More on this in the methodology section.

The code for the simulation is written in the Fluid.cpp class as the following functions:

1. Diffuse(): This function mainly serves as a wrapper for LinearSolve().
2. LinearSolve(): This function averages values across all non-border pixels and is used to diffuse x and y velocity, as well as fluid density.

3. SetBoundary(): This function is responsible for nullifying diffusion effects at the border of the simulation space, making it so fluid currents dampen as they bump into the walls.
4. Project(): The functionality of Project() is not explained in the comments, and the code does not make it obvious due to its complexity.
5. Advect(): This function makes the fluid (or more specifically its density) actually move based on its velocity values.

As for function call hierarchy: Diffuse() and Project() call LinearSolve() and LinearSolve(), Project() and Advect() all call SetBoundary(). This detail will come into play when trying to iteratively port each function to the GPU.

Optimization Methodology

As mentioned in the introduction, we decided on porting the simulation to the GPU before even starting, as we had no prior experience with GPU programming, making this a good learning opportunity, and grid-based simulations are perfect candidates for GPU computations. However, when it came to actually deciding which parts of the codebase to port, we obviously needed to profile the application to determine where to focus our efforts most effectively. Each time we profiled, we took the apparent bottleneck function and optimized it, before profiling again. What follows is a series of paragraphs explaining the steps we took in chronological order based on what we found from profiling.

LinearSolve() and SetBoundary()

The first time we profiled immediately showed that LinearSolve() was by far the slowest part of the software containing 16 iterations of an N x N for-loop, where N is one dimension of the grid. This seemed like a perfect use-case for GPGPU as all N×N operations could be executed independently of each other without fear of race-conditions (this isn't entirely true, more on this in the discussion section) or slowdown due to the prevention thereof. This would still have to be repeated 16 times as the calculation of each cell relies on the previously updated values of their neighbors. After successfully implementing a kernel for this operation and moving the work to the GPU, we noticed that our performance had actually dropped to roughly 0.25x of the baseline. Creating a new buffer and reading it back for every one of the 16 iterations meant that data transfer to and from the GPU introduced so much overhead that the speedup of performing the calculations on the GPU was no longer worth it. This is a pattern we see repeatedly throughout the process; it's only towards the very end that performance started to rise significantly. It was evident that as much work as possible should be done on the GPU without intermediate data transfers. To that end, we also had to port SetBoundary() to the GPU, as LinearSolve() called this function after every iteration, which is why we had to repeatedly read and write the buffer in the first place. We created a duplicate of the SetBoundary function that would instead take a buffer as input and which ran on the GPU. We then replaced the SetBoundary() call in LinearSolve() with the GPU version which allows us to hoist the buffer

creation and loading outside of the for-loop. This was the point at which we saw our first actual speedup to roughly 1.25x the baseline.

Project() and Advect()

Upon profiling after the previous changes, we saw that Project was now the more costly function, and we decided to port it to the GPU as well. We hypothesized that having to read back the buffers from Project() before calling Advect() and writing them again afterwards would be slow, but we didn't want to port Advect() yet; premature optimization is the root of all evil, after all. We began by porting Project() but quickly realized an issue with this. The Project() function begins with an N×N for-loop, then makes several calls to LinearSolve() and SetBoundary(), and then performs another N×N for-loop. As we had not yet been able to hoist buffer creation out of LinearSolve() due to architectural constraints, we were forced to refactor Project() into the following structure:

1. Load data into buffers and perform the kernel replacement of the first for-loop
2. Read out all the data from the buffers
3. Call LinearSolve(), which itself also loads and reads the data in and out of buffers
4. Load data into buffers and perform the kernel replacement of the second for-loop
5. Read out all the data from the buffers

We accepted this as a temporary constraint until we would be able to hoist up all buffer creation but we were pleasantly surprised to still see a performance increase to just under 1.5x of the baseline. Profiling again showed what we were hoping to see: Advect() was now the bottleneck, so it was justified to port it as well. Initially this seemed like it would be troublesome, considering the many casts and lines of conditional code it contained. Fortunately, when taking a closer look, we realized that many of the casts were unnecessary and could be circumvented in one way or another. The conditional code could also be tweaked which allowed us to get rid of a number of unnecessary declarations and assignments.

At this point we could move all components that make up the fluid update loop to the GPU, meaning buffer creation could be hoisted out of all the functions and into the main update function, greatly decreasing the amount of overhead caused by the many data transfers. All buffers were written just once, then all kernels were executed, keeping data in GPU memory for the entirety of an update cycle, after which all data could be read out again.

SIMD

After considering and rejecting some more optimizations (explained in the discussion), we decided to leave the simulation code as it was. The bottleneck was now the rendering of the simulation. The base project uses the PixelGameEngine library for this, which made it hard to change how things were done. Hijacking the rendering and doing it ourselves would also be a big undertaking, since the PixelGameEngine library was also responsible for creating the window, and when rendering using the GPU, it is best not to read back the updated array, but to directly set the rendertarget on the GPU. Both of these tasks were new to us, and we didn't have as much time as we would have liked, so we decided it would be more efficient to stick to what we know and implement one last, lesser but more straightforward optimization than porting

the rendering to the GPU: SIMD. The draw call in FluidSimulation::OnUserUpdate() used a clamp and setpixel function for every cell in the simulation array. This seemed to be quite slow. We added SIMD to the loop, handling four cells at a time instead, with a small loop at the end to handle extra cells in case the array was not of a size divisible by four. Our implementation is not the most efficient, creating a temporary array for every four entries so that the integers could be read as individual bytes. This probably could have been optimized further, but it was at this point that we realized that the rendering bottleneck displayed in the profiler was actually probably the OnUserUpdate() function waiting for its call to the fluid update to complete. The for loop to which we were applying SIMD was thus not that much of a bottleneck, the thought of which was reinforced by the rather mediocre performance improvement that the optimization resulted in. With the deadline coming up, this is where we decided to end the optimization process, leaving us with the final results as mentioned below.

Results

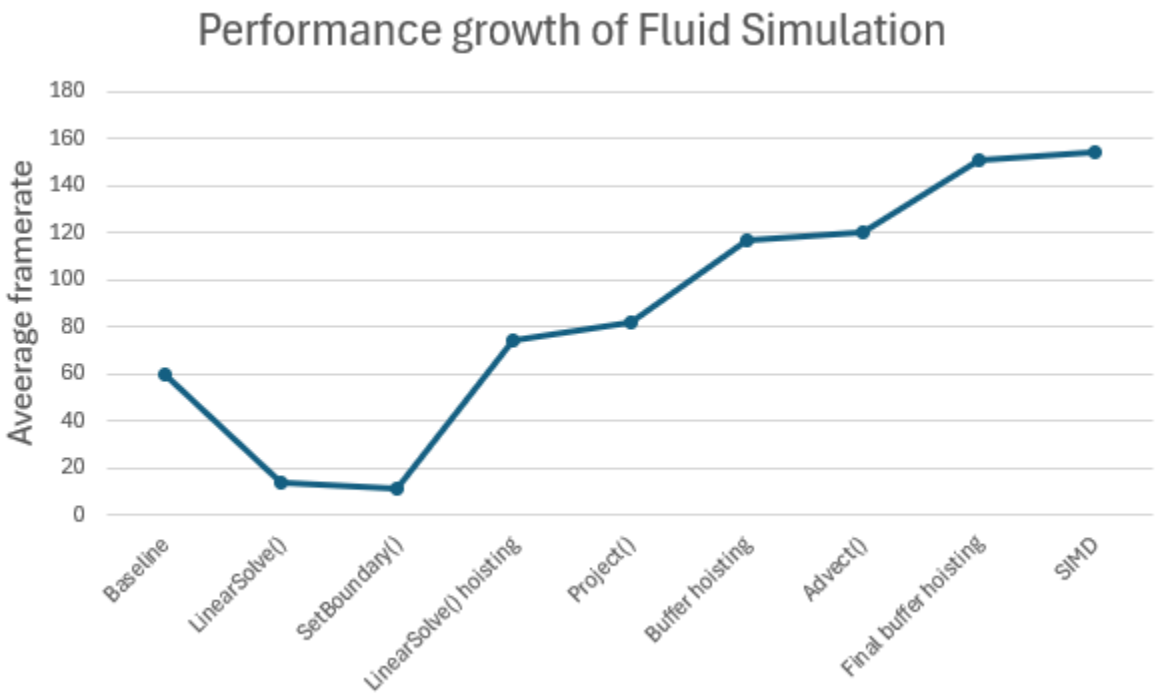
Since the base project already included a framerate display, we decided to take that as our performance measure. Below is a table containing the framerate of the program at various stages of optimization. All measurements were taken in short succession on the same system and in release mode to avoid external interferences that may affect results. Measurements were done on a laptop with the following specifications:

- CPU: AMD Ryzen 5 4600H
- GPU: Geforce GTX 1650
- RAM: 8GB ddr4

Progress stage	Average	Min	Max
Baseline	60	60	62
LinearSolve() on GPU	14	12	17
SetBoundary() on GPU	11	10	14
Hoisted buffer creation out of loop, minimizing data transfers for LinearSolve()	74	67	81
Project() on GPU	82	77	91
Buffer creation hoisted to update	117	112	120
Advect() on GPU	120	115	129

Advect() buffers hoisted to update	151	143	158
SIMD	154	149	163

The following graph shows the average framerate from the table above in a more easily digestible format.



Greater theoretical speedup

As a final note on the results, keep in mind that this performance is based on a 160 x 160 grid and 16 diffusion iterations per frame, with all the overhead of reading and writing buffers on each of those frames. The actual calculations themselves are performed much more quickly than the x2.57 speedup would imply, but the overhead of each update holds them down. In theory, the current codebase is much more scalable performance-wise, and should be able to handle much larger grid sizes (15 to 60 fps for 320 x 320 grid = x4 speedup) or more diffusion iterations (18 to 90 fps for 64 iterations = x5 speedup) without slowing down nearly as much as the base code. At that point, it would of course be well worth it to always keep the data on the GPU without any data transfers, but as explained in the discussion, we did not get around to this.

Discussion

Left out Optimizations

Some optimizations that we considered were left out. The first of these was to hoist buffer reaction up even higher, out of the update loop, so that there could be even less GPU data transfers. Unfortunately there were various calculations outside of the simulation code that needed the data every frame, such as the mouse input code. It might have been possible to port this to the GPU as well, sending over only mouse coordinates and clicking information each frame, but considering how small a part the simulation code had become of total runtime, with rendering now being the bottleneck, this would probably not have had much impact, so we opted to avoid premature optimization and leave it as is. This is also the reason why the second left out optimization was abandoned: Hoisting kernel initialization out of their respective functions (LinearSolve(), SetBoundary(), etc.) and into the constructor of Fluid.cpp. This might have improved performance slightly, but considering that the simulation code was no longer the bottleneck, it didn't seem worth it.

Erroneous functionality in the base code

As mentioned in the introduction, we realized that baderouaich probably made a mistake while implementing the Lattice Boltzmann method. In the LinearSolve() function, each cell sets its own value based on its neighbors, but this is done in situ instead of in a new array, so the next cell reads the updated value. The result is that every value in the grid will have a value based at least partially on all its neighbors on the left in one iteration, while not being influenced by any of its neighbors on the right, save the one right besides them. The same thing goes for the y-axis. It may be that with enough iterations this effect becomes negligible, so it could be a deliberate choice to ignore it for the sake of performance, but it is technically incorrect. Our ported code on the GPU has this same issue but in another order. Cells are handled in parallel, making it inconsistent which cells "go first", and whether they overwrite their own values before their neighbors read them. This means that we technically slightly changed the functionality of the simulation, as it is no longer deterministic. However, in practice, there is no noticeable difference in the behavior of the fluid, and if anything it is an improvement, as random updating of cells means there is no longer a sort of "priority" for cells in the top left when it comes to fluid diffusion.

Notes

- It is recommended to run the project in release mode. In the base project, changing from debug to release mode increased our fps from 1 to about 60, and in our final version it's the difference between 30 and 150 fps.
- We had issues getting the code to run on AMD hardware due to OpenCL specifics, so either an Intel CPU or GPU or an Nvidia GPU is recommended to run the application.