Stable Fluid Simulation

David Perrone and Jacob Afonso

Introduction

Particle simulation was a major part of the class and industry used for simulation real life physics. Fluid simulation specifically is widely used in certain scientific applications as well as video games. We decided to implement a parallelized version of a stable fluid simulation based on the serial version described by Jos Stam in “Real-Time Fluid Dynamics for Games”. We also wanted to render his simulation which we did through OpenGL.This simulation is based off of the Navier-Stokes equations, which are majorly famous functions describing the behavior of fluids.The simulation models adding a “dye”/density with some velocity to a point in a box filled with an incompressible fluid. The fluid being incompressible means that its density cannot change, so that is why when adding a dye, there is movement throughout the fluid box.

There are 3 main components to the physics of the simulation: diffusion, advection, and projection. Diffusion simulates the effect of a dye spreading out when put into a fluid. This is done by each particle passing some of its density and velocity to its 4 cardinal neighbors.

Advection: simulates the movement of a dye along the velocity field of a cell. Basically, if a liquid is moving it will carry other things with it.

Diffusion: is defined as the intermingling of substances by the natural movement of their particles. In the case of this simulation, diffusion represents how a dye could move through fluid even if the fluid is perfectly still. Some fluids might spread faster than others like molasses vs rubbing alcohol. Diffusion represents the viscosity of the liquid and how resistant it is to momentum.

Projection: is used to keep the amount of fluid in the box constant. This is an incompressible fluid simulation - meaning that the fluid going out of and into a box must be equal.

These functions are all brought together to simulate a single step in time, which we can also define as our dt. Each time step diffuses the x and y velocities, then projects based on these new velocities. Then the advection occurs on both the x and y velocities and again projects on those updated velocities. Then finally the density undergoes diffusion then advection as well.

Related Work

As mentioned, the inspiration for our project came from the work of Jos Stam. He created the original serial functions that implemented diffusion, advection, and projection, as well as the way these functions will carry out across the fluids density and velocities.

Another piece of related work was carried out by Mike Ash. He extended Jos Stam's simulation to 3-dimensions and also put together a simplified explanation of the functions themselves which made them easier to understand.

There was also a youtube video by The Coding Train, where he rendered the 2D simulation using JavaScript. This gave us an idea of how it would look when we rendered it ourselves and another perspective on what the functions themselves did.

Parallelization

The three main functions in our simulation were for diffusion, projection, and advection. They used two helper functions lin_solve and set_bnd which are used to solve a sparse linear system and to set the boundaries of the square. The original implementation was done in C++ and we chose to implement our version using cuda with C++. We converted diffuse, advect, and project into global functions and made lin_solve and set_bnd device functions.

Our approach for parallelization was to replace loops in these functions with kernel calls. We first defined a certain number of threads and then used that and the size of the square to determine how many blocks would be needed. This would look something like:

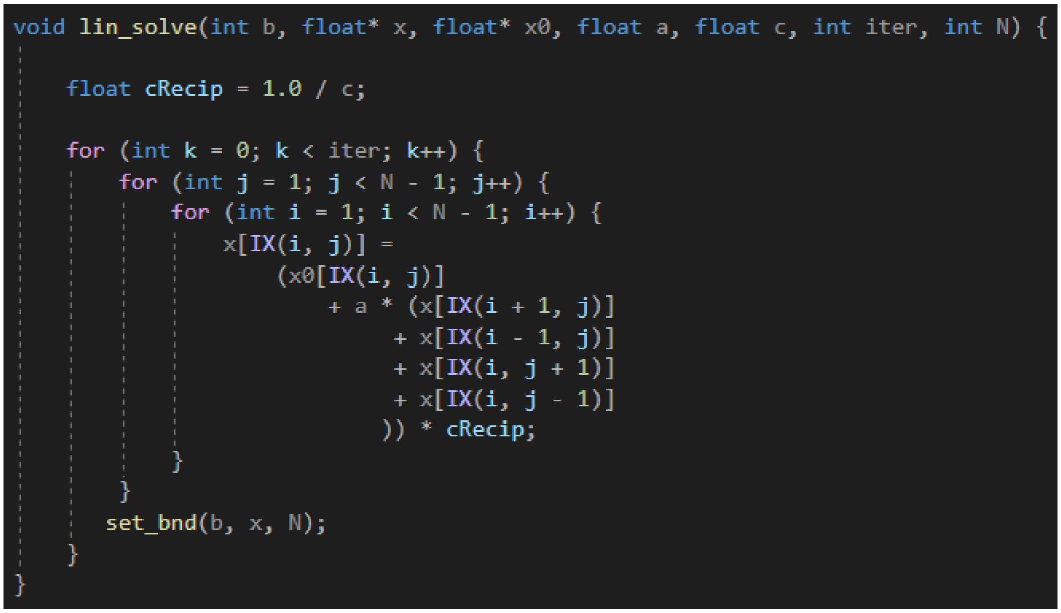
```
#define NUM_THREADS 256

Int num_blks = (N * N + NUM_THREADS - 1) / NUM_THREADS;
```

After that, we would call a kernel and specify how many blocks and threads are to be used:

```
diffuse_gpu <<<blks, NUM_THREADS>>>  (1, sq->Vx0, sq->Vx, visc, dt, 4, N);
```

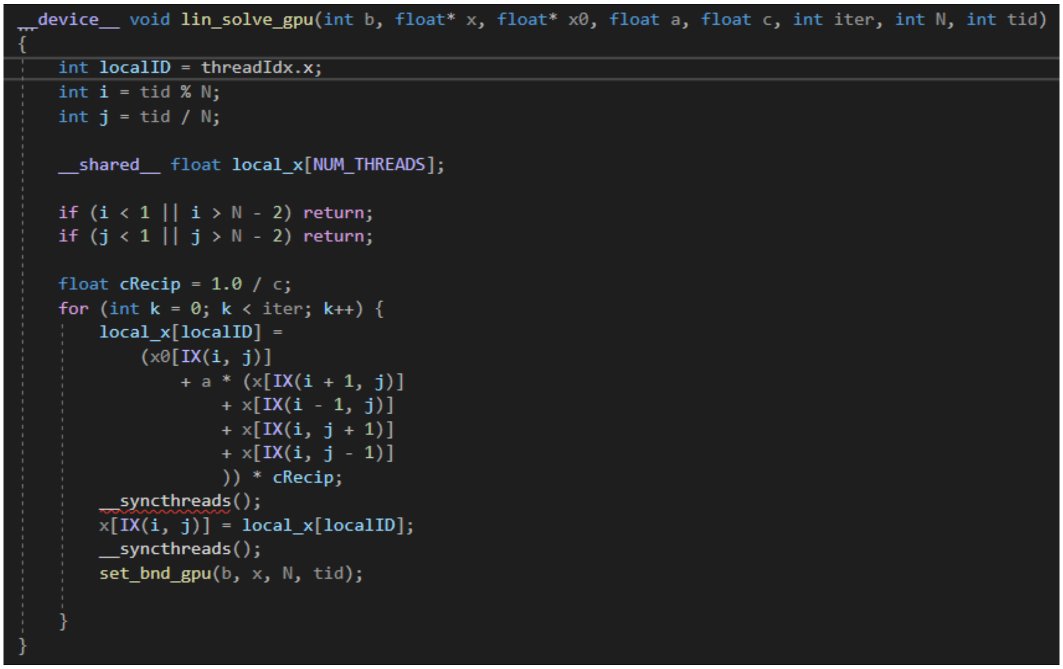
This was the same for all of our main 3 physics functions, diffuse, advect, and project. Where there was looping in these functions, we replaced them with a single update based on an i and j calculated from the thread id.

These functions call helper functions, lin_solve and set_bnd. These functions needed more major changes to parallelize. These were implemented as device only functions that were only called from within the diffuse, advect, and project kernels. In these, we pass the threadId into the function call as a parameter. Here is what the original lin_solve function looked like:



```
void lin_solve(int b, float* x, float* x0, float a, float c, int iter, int N) {

    float cRecip = 1.0 / c;

    for (int k = 0; k < iter; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
                x[IX(i, j)] =
                    (x0[IX(i, j)]
                        + a * (x[IX(i + 1, j)]
                             + x[IX(i - 1, j)]
                             + x[IX(i, j + 1)]
                             + x[IX(i, j - 1)]
                            )) * cRecip;
            }
        }
        set_bnd(b, x, N);
    }
}
```

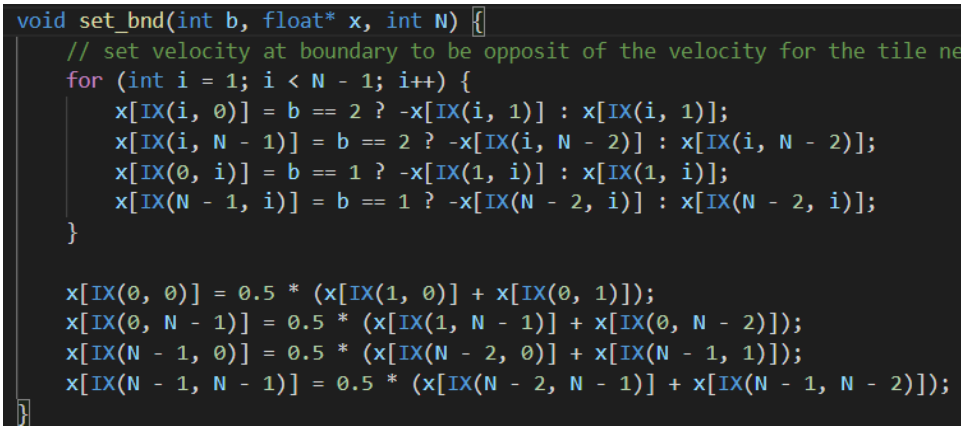
This function contains 3 loops. The outermost loop is not replaceable because it specifies how many times the linear system should be solved, essentially being our level of approximation akin to approximating integrals using a number of rectangles. The inner two loops however can be parallelized. Instead of iterating through one by one and solving each cell we would call it from within a kernel corresponding to a single particle. The lin_solve function is now defined as:



```
__device__ void lin_solve_gpu(int b, float* x, float* x0, float a, float c, int iter, int N, int tid)
{
    int localID = threadIdx.x;
    int i = tid % N;
    int j = tid / N;

    __shared__ float local_x[NUM_THREADS];

    if (i < 1 || i > N - 2) return;
    if (j < 1 || j > N - 2) return;

    float cRecip = 1.0 / c;
    for (int k = 0; k < iter; k++) {
        local_x[localID] =
            (x0[IX(i, j)]
                + a * (x[IX(i + 1, j)]
                    + x[IX(i - 1, j)]
                    + x[IX(i, j + 1)]
                    + x[IX(i, j - 1)]
                    )) * cRecip;
        __syncthreads();
        x[IX(i, j)] = local_x[localID];
        __syncthreads();
        set_bnd_gpu(b, x, N, tid);

    }
}
```

The inner loops are now gone and tid (thread id) is used to index into the square. We also added a shared array for each block so that the kernels would read from global memory, write into shared memory, synchronize and then write the results back to global memory. This is because overwriting elements of the x array in any order that we could not control would mess up the calculations of other points in the array. The temporary shared array is not a perfect solution, but ended up giving us just about the same behavior as the serial version had.

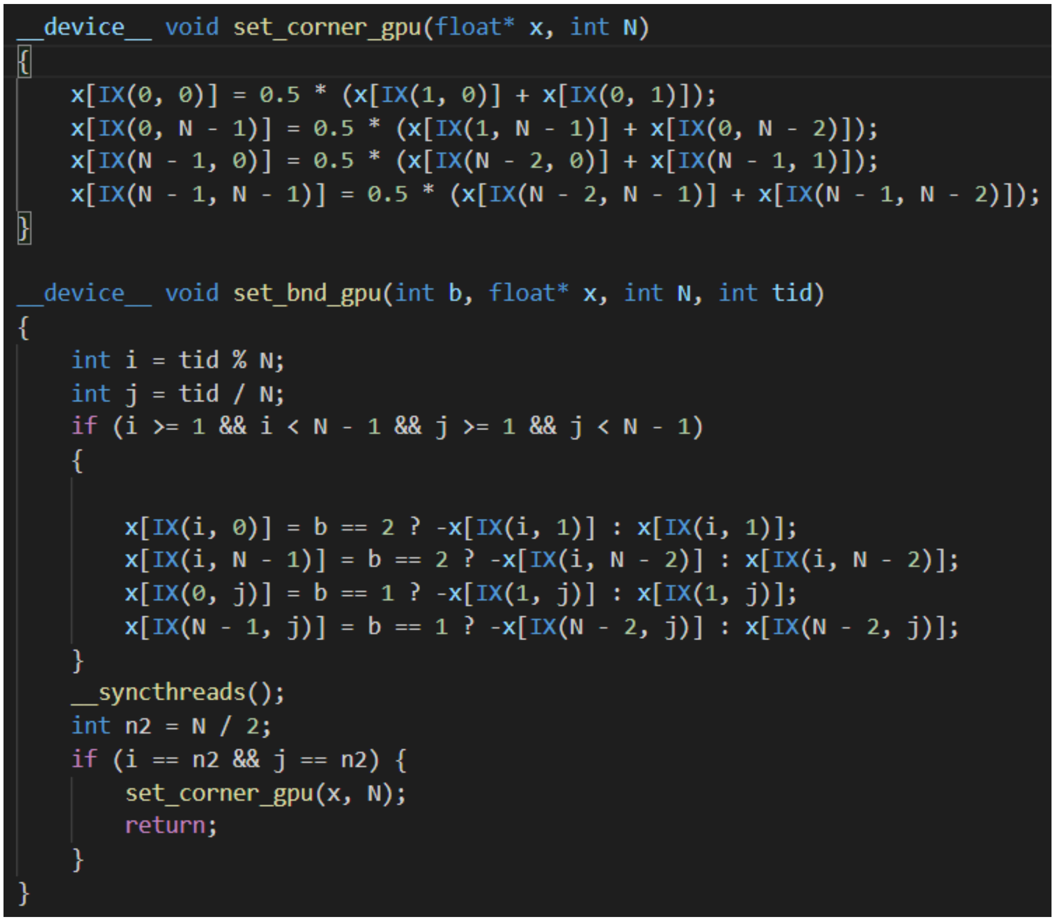
Another important point is that the i and j loops iterate from 1 to N-2, to essentially cover every particle not on the boundary. The boundary cases are handled by the set_bnd() functions. This function essentially sets the density/velocity (depending on which array ‘x’ is in the call) to be the opposite of its 1 direct neighbor. This gave the effect of bouncing off the wall. The serial version of this function went through every boundary point and did this update, then for the corners, it was set to the average of its 2 neighbors. That looked like this:



```
void set_bnd(int b, float* x, int N) {
    // set velocity at boundary to be opposit of the velocity for the tile ne
    for (int i = 1; i < N - 1; i++) {
        x[IX(i, 0)] = b == 2 ? -x[IX(i, 1)] : x[IX(i, 1)];
        x[IX(i, N - 1)] = b == 2 ? -x[IX(i, N - 2)] : x[IX(i, N - 2)];
        x[IX(0, i)] = b == 1 ? -x[IX(1, i)] : x[IX(1, i)];
        x[IX(N - 1, i)] = b == 1 ? -x[IX(N - 2, i)] : x[IX(N - 2, i)];
    }

    x[IX(0, 0)] = 0.5 * (x[IX(1, 0)] + x[IX(0, 1)]);
    x[IX(0, N - 1)] = 0.5 * (x[IX(1, N - 1)] + x[IX(0, N - 2)]);
    x[IX(N - 1, 0)] = 0.5 * (x[IX(N - 2, 0)] + x[IX(N - 1, 1)]);
    x[IX(N - 1, N - 1)] = 0.5 * (x[IX(N - 2, N - 1)] + x[IX(N - 1, N - 2)]);
}
```

The loop corresponds to the boundaries, and the 4 lines outside are for each corner.

To parallelize this, we replaced the loop with thread indexing. This meant we calculated i and j based on the tid and then only did a calculation if the i and j were both between 1 and N-2 inclusive. Then for the corners, we wrapped that in its own function that was only called by a single thread. This ended up looking like this:

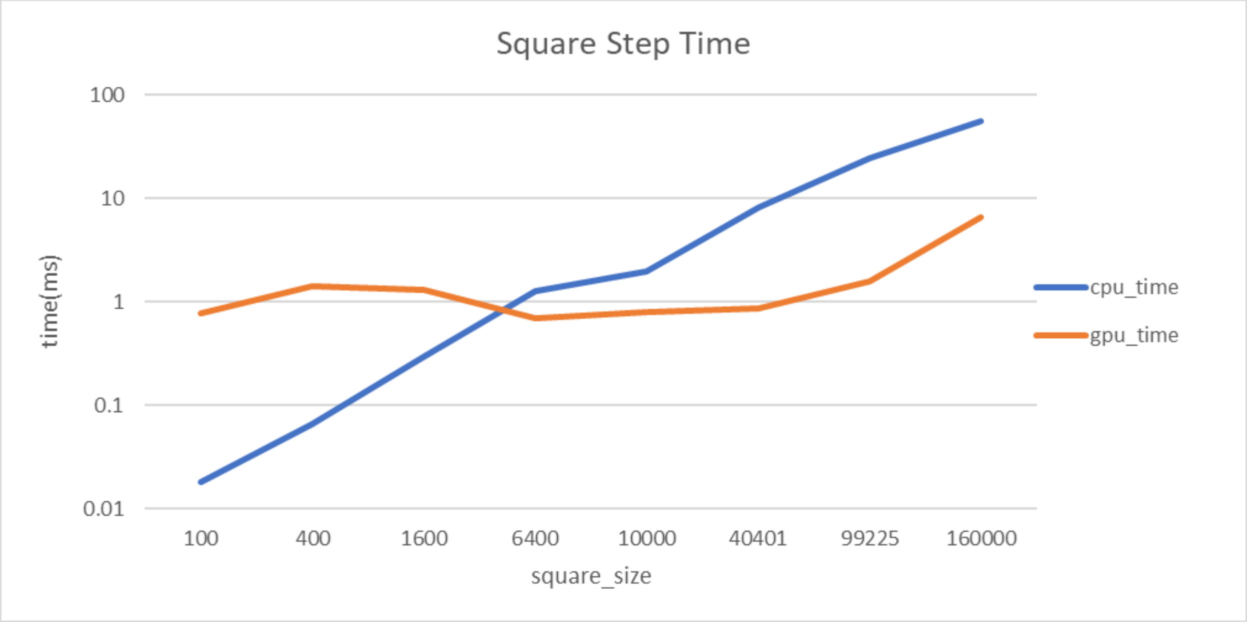


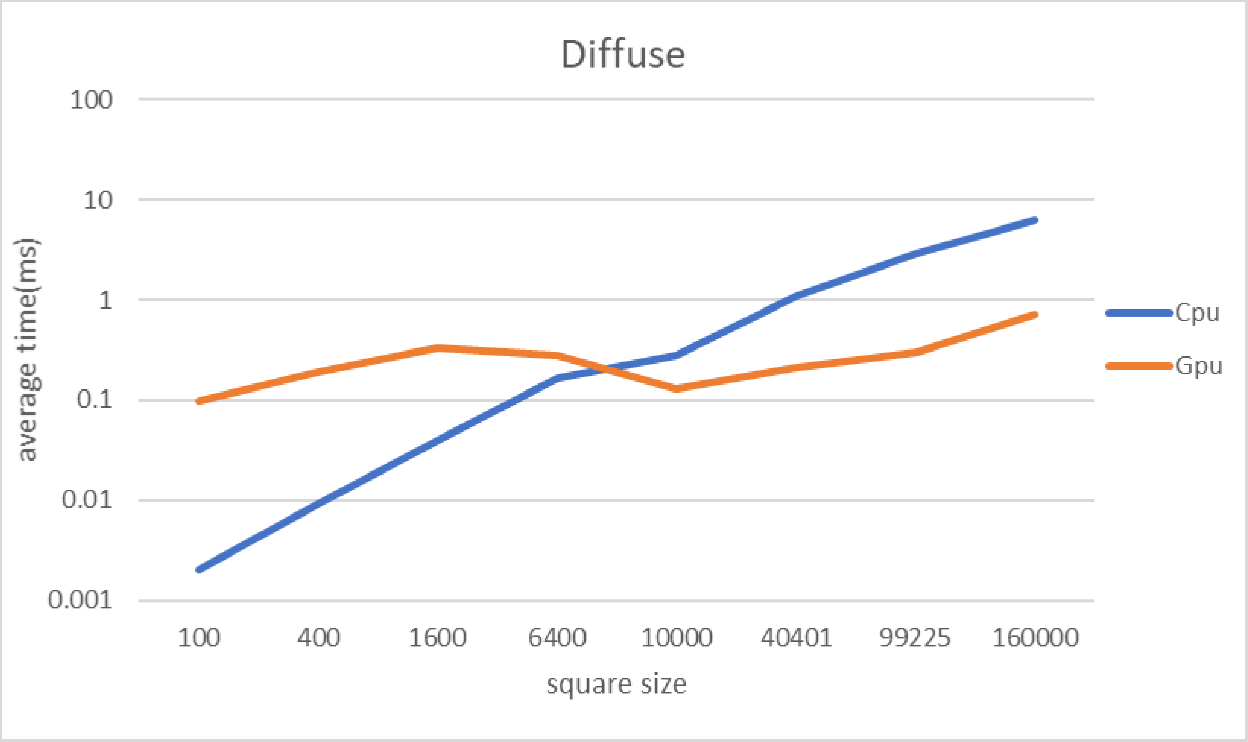
```
__device__ void set_corner_gpu(float* x, int N)
{
    x[IX(0, 0)] = 0.5 * (x[IX(1, 0)] + x[IX(0, 1)]);
    x[IX(0, N - 1)] = 0.5 * (x[IX(1, N - 1)] + x[IX(0, N - 2)]);
    x[IX(N - 1, 0)] = 0.5 * (x[IX(N - 2, 0)] + x[IX(N - 1, 1)]);
    x[IX(N - 1, N - 1)] = 0.5 * (x[IX(N - 2, N - 1)] + x[IX(N - 1, N - 2)]);
}

__device__ void set_bnd_gpu(int b, float* x, int N, int tid)
{
    int i = tid % N;
    int j = tid / N;
    if (i >= 1 && i < N - 1 && j >= 1 && j < N - 1)
    {

        x[IX(i, 0)] = b == 2 ? -x[IX(i, 1)] : x[IX(i, 1)];
        x[IX(i, N - 1)] = b == 2 ? -x[IX(i, N - 2)] : x[IX(i, N - 2)];
        x[IX(0, j)] = b == 1 ? -x[IX(1, j)] : x[IX(1, j)];
        x[IX(N - 1, j)] = b == 1 ? -x[IX(N - 2, j)] : x[IX(N - 2, j)];
    }
    __syncthreads();
    int n2 = N / 2;
    if (i == n2 && j == n2) {
        set_corner_gpu(x, N);
        return;
    }
}
```

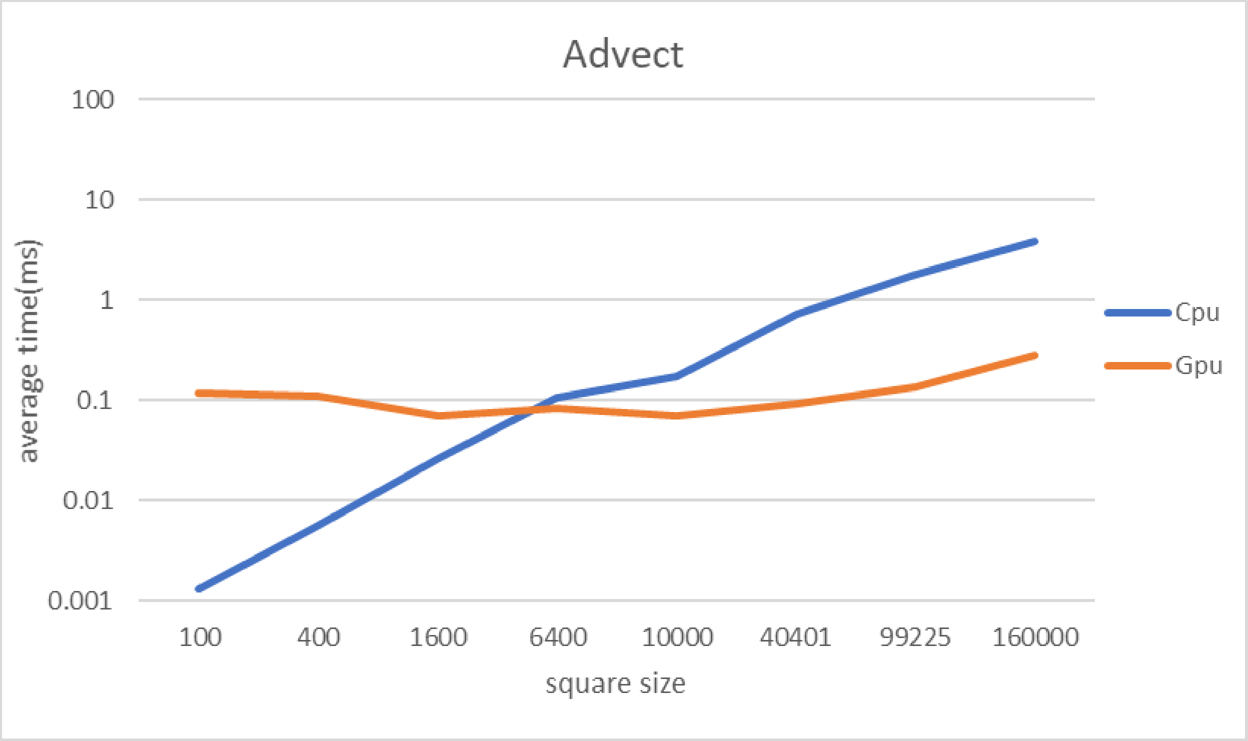
In terms of synchronization, we had to use a good amount of synthreads() calls throughout the program. In between each overall kernel, we synchronized the threads to ensure that each thread was done with their calculations before moving onto the next kernel call. We also needed synchronization within the kernels and device calls themselves. These all came before and after calling the device functions lin_solve() and set_bnd(). We needed to ensure that all threads finished before entering these functions and after completing these functions since the calculations within these were dependent on other threads. With the calculations within the kernels themselves, we did not have a need for synchronization as these calculations read from one array and wrote into a different, so there were no inter-thread dependencies.

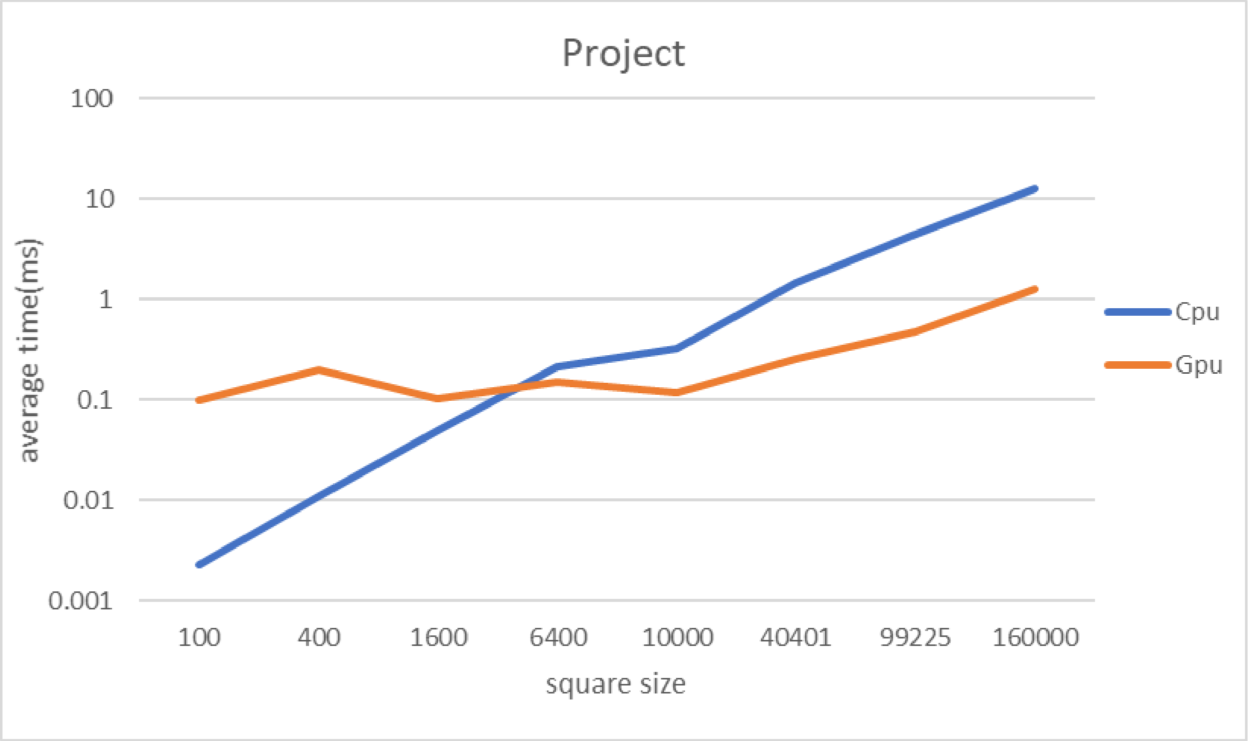
Performance Analysis

To measure the performance of our parallelized version, we compared the serial times with the cuda times for each function(advect, diffuse, and project) as well as the overall time step which calls all these functions as described in the introduction to simulate a single step forward in time. We also used an increasing amount of particles in the simulation, which we denoted as the square size. The results can be seen in the following plots:



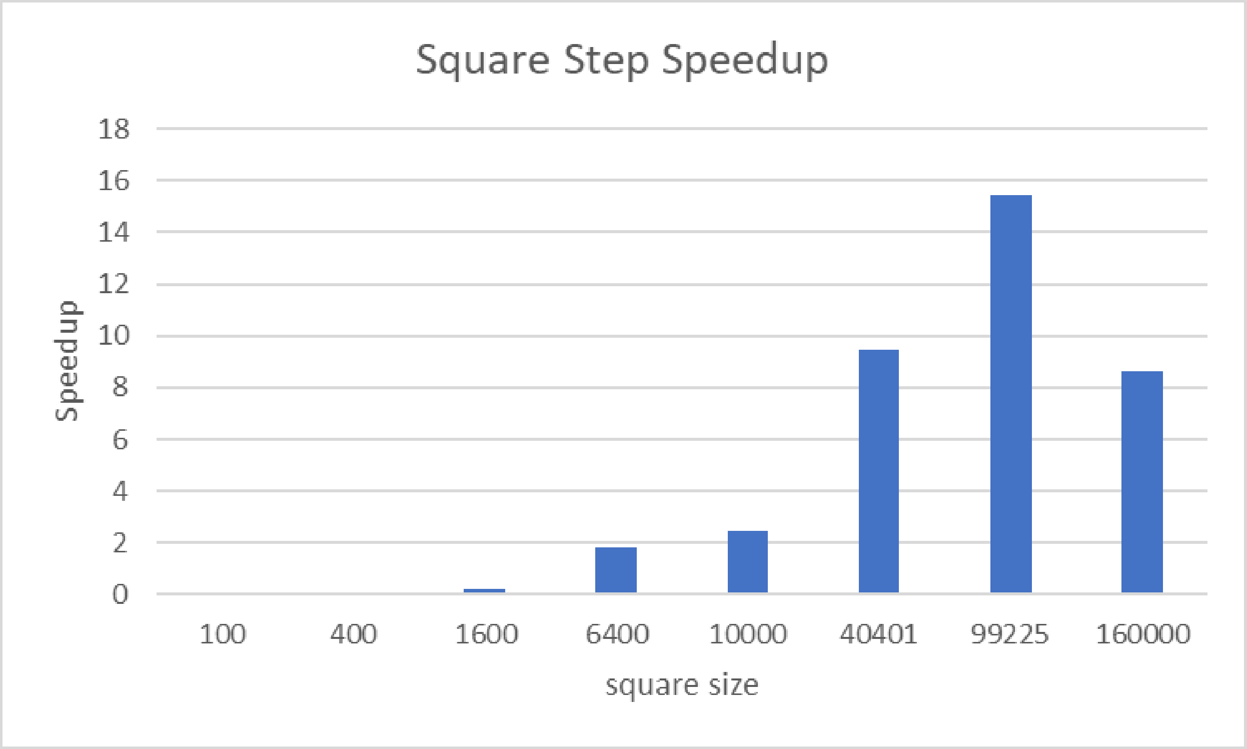

Diffuse
100
10
1
0.1
0.01
0.001
average time(ms)
100
400
1600
6400
10000
40401
99225
160000
square size
Cpu
Gpu

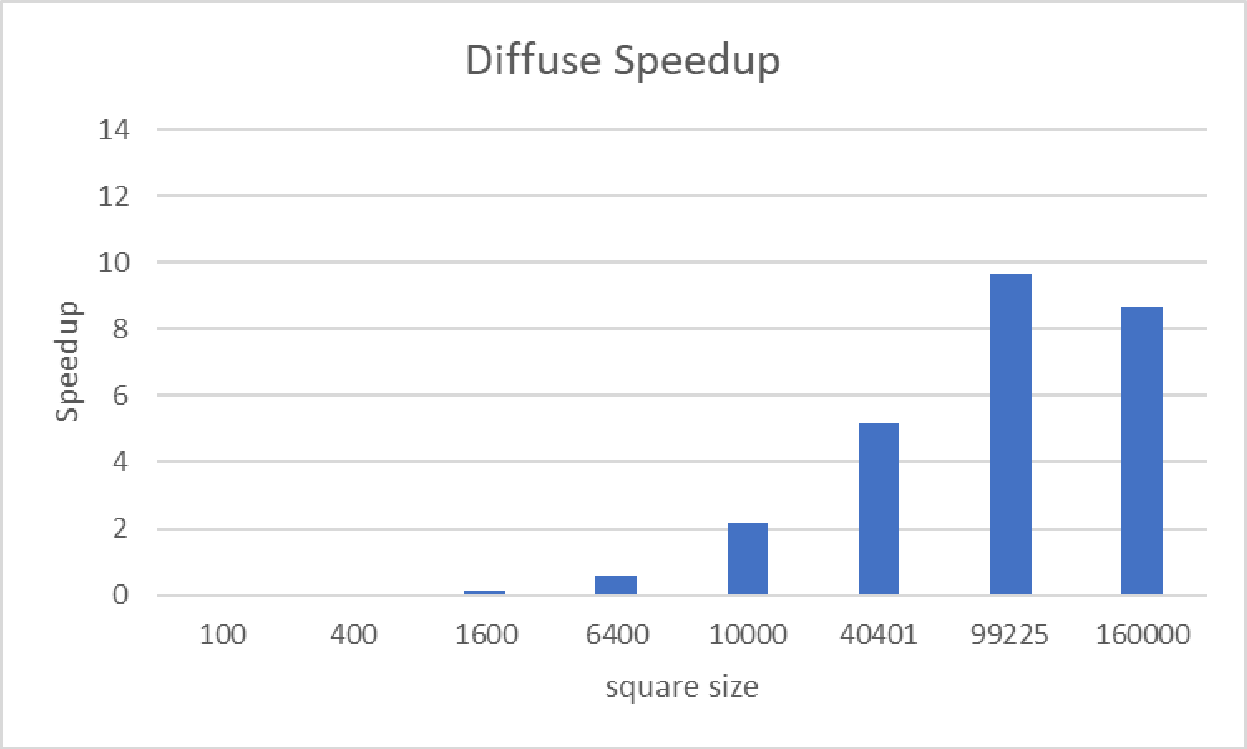

Advect
100
10
1
0.1
0.01
0.001
average time(ms)
100
400
1600
6400
10000
40401
99225
160000
square size
Cpu
Gpu

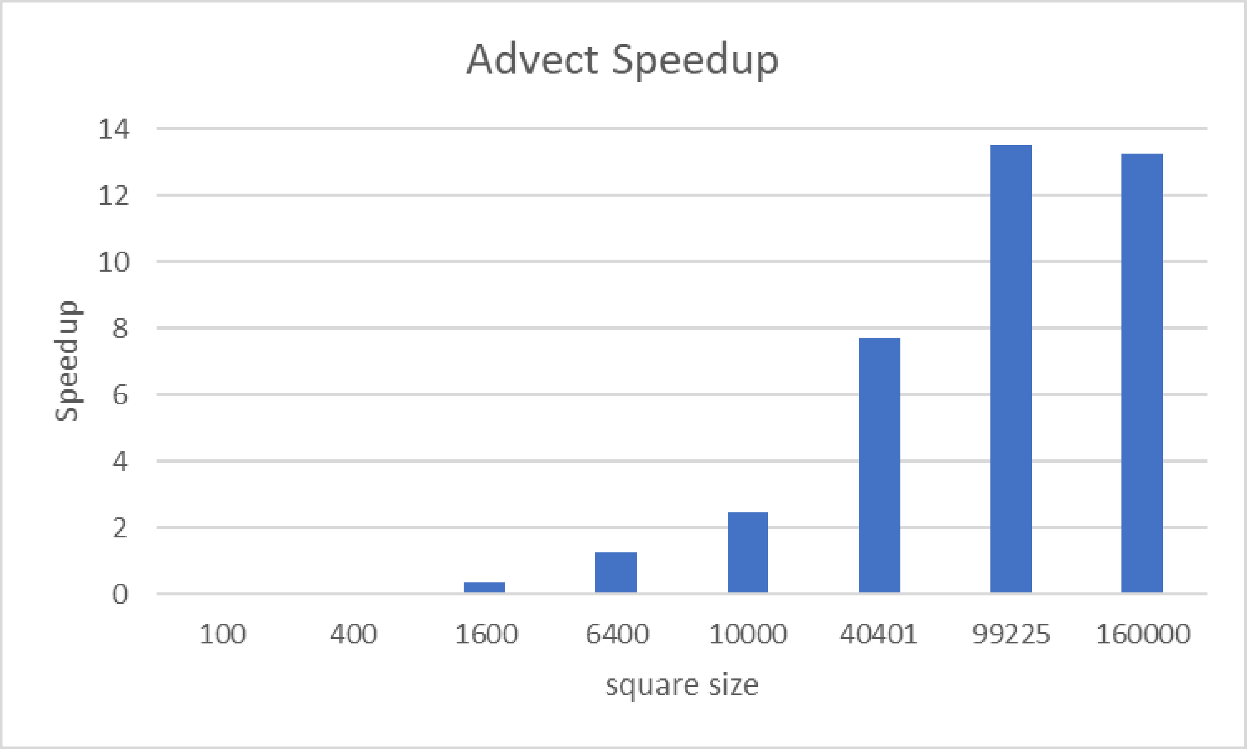


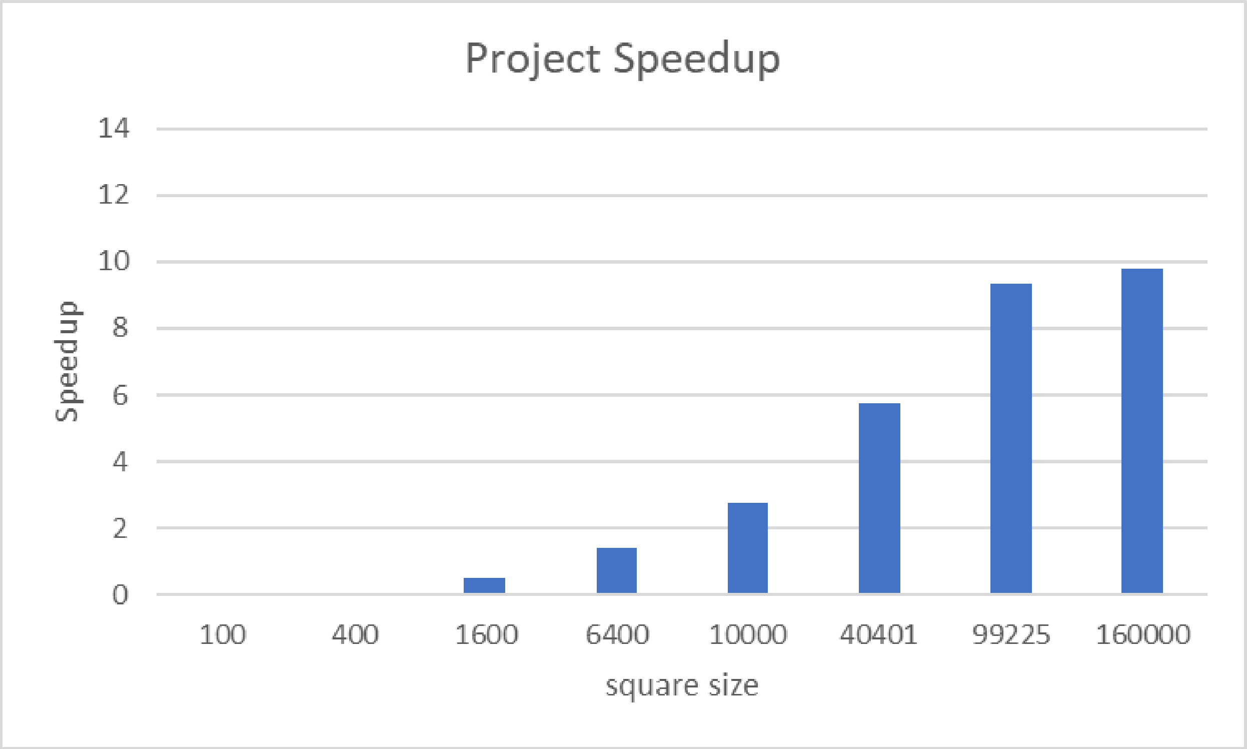
The results of these timing experiments showed us that for smaller square sizes, the cpu ran faster then the gpu, but once the square size reached around 6400 particles, the gpu outperformed the cpu. This was a pretty expected result, as the overhead of using cuda and the gpu was not worth it for smaller amounts of particles, but became faster for larger amounts since we could get so much parallelism. Also, the gpu performed very close to constant across the increasing square sizes, with the time for square step staying around 1 ms and the function times staying around .1 ms. Only for the last square size of 160,000 did we start to see the gpu times drastically increase. For the cpu, the algorithm performed exponentially(note the log scale time) with the time increasing drastically as the square size increased.

For the speedups themselves, they are plotted below:


Square Step Speedup
Speedup
18
16
14
12
10
8
6
4
2
0
100
400
1600
6400
10000
40401
99225
160000
square size


Diffuse Speedup
Speedup
14
12
10
8
6
4
2
0
100
400
1600
6400
10000
40401
99225
160000
square size





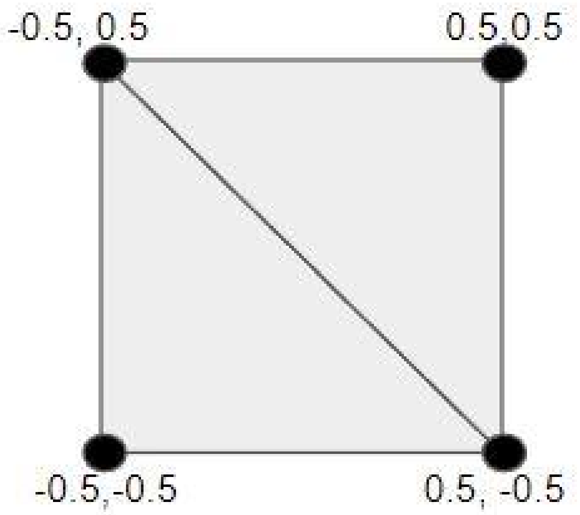
These plots show how for the overall step, we reach a peak speedup of about 15x the serial for 99,225 particles. Then once we extend past that to 160,000 particles, there is presumably not enough resources to parallelize as well, so the speedup starts to decline. This is also the case with the individual functions themselves, to a lesser extent. The speedup starts to decline at 160,000 particles, besides for the project, but the growth to that amount is significantly less than the previous jump there.

Rendering

We decided to render the results of our cuda calculations using opengl. A major part of graphics programming in both Cuda and OpenGL is limiting the data transfers between CPU and GPU. Cuda can perform the calculations but we needed to find a way to transfer that data to openGL for the draw calls. Copying from GPU to CPU to GPU to transfer the data is not an efficient way to go about this so we looked into cuda-opengl interoperability. This would allow Cuda to write results of the calculations directly into an OpenGL buffer and save on the costly copies.

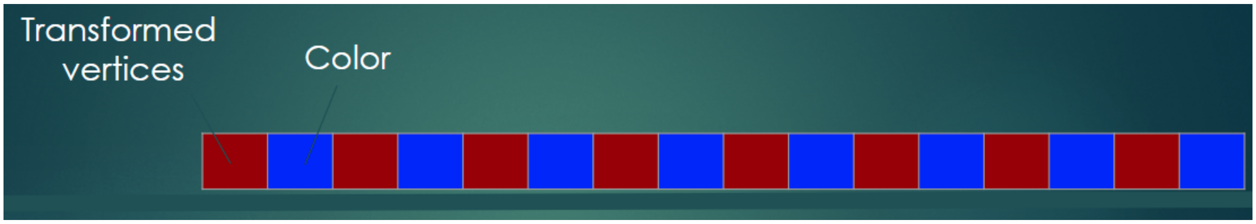
Vertices

Shapes in openGL (and most rendering) are represented in triangles. The 2d Fluid square is represented as a 2d array of smaller squares which are made of two triangles each. Here is an example of a typical representation of a square:

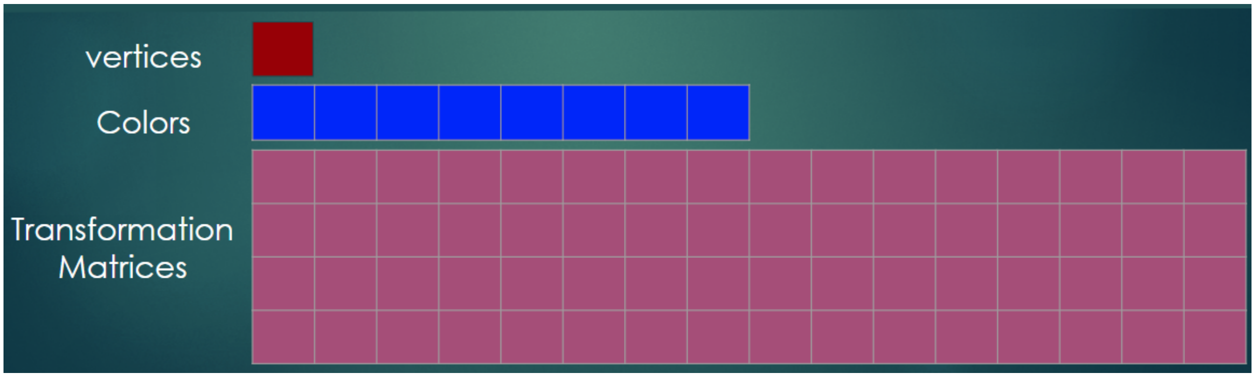


The basic idea here is to multiply each of these vertices with a transformation matrix to move it around the screen. Colors for the square would be represented as a vector of 4 floats (rgba) and would need to be updated with the density values calculated by the Stable Fluids simulation. Rather than perform computations for either of these and cpu and transfer data to OpenGL to render we decided to rely on Cuda for both the matrix multiplication and the color updates. This saved on the costly copies and allowed us to parallelize the communication between Cuda and OpenGl by using Cuda kernels to update matrices and color vectors.

We then needed to find a way to optimize the OpenGL draw calls. It wouldn’t be much help if we performed the square steps in parallel and then had to draw each rectangle one at a time. The two popular ways of minimizing draw calls is to use batch rendering or instanced rendering. We had found a good implementation of batch rendering online but did not think it would be easy to parallelize. The algorithm is essentially to build up a buffer of each object that you want to draw and then when the buffer is full or you run out of objects to have OpenGL draw the objects in one draw call. The memory layout of such a batched draw call would look like this:

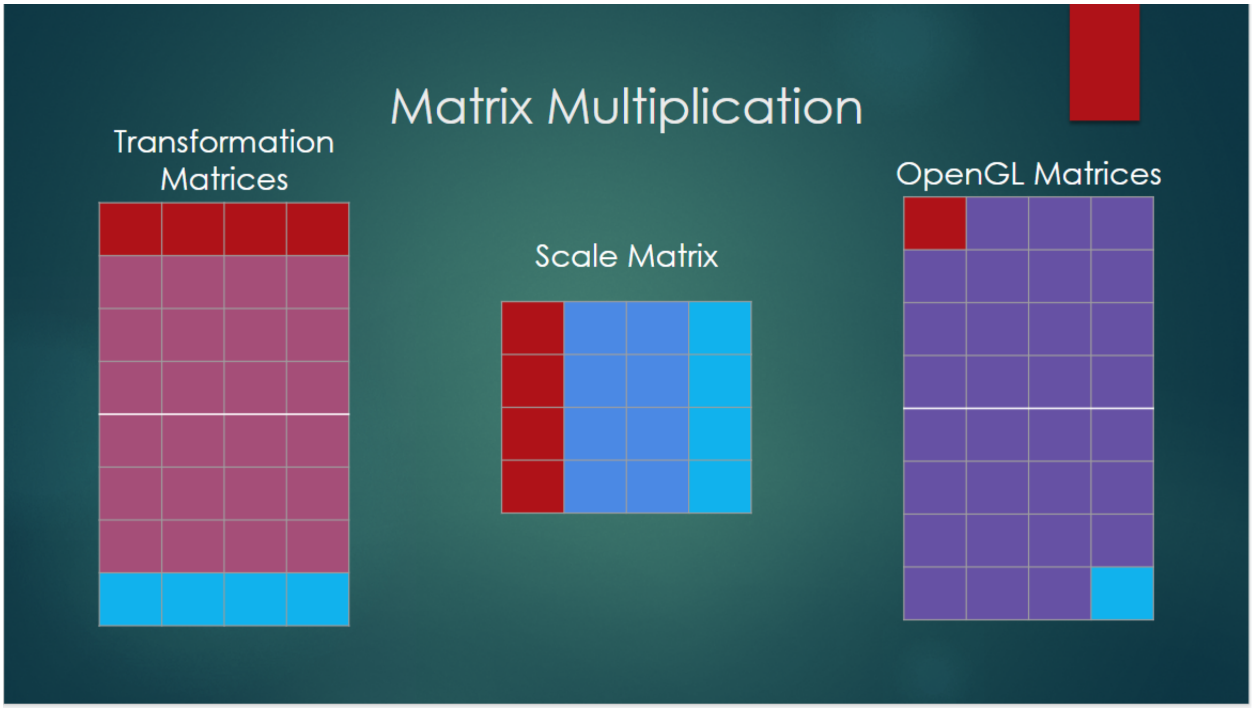


The alternative to batch rendering was instanced rendering. This required supplying OpenGL with one set of vertices representing a square, the number of squares you want drawn, and a buffer of transformation matrices for each "instance" of the square. OpenGL can then draw thousands of squares in one draw call. This method also meant that we could separate vertices, matrices, and color vectors into separate memory buffers which look more like this:



This memory layout would work much better with Cuda-OpenGL interoperability. The basics of how this works is that Cuda would register a graphics resource object with an opengl buffer and can then query OpenGL for a pointer to where that buffer is stored on the GPU. Cuda can then use that buffer like any other device buffer and write data directly into it. We would not need to worry about interleaving vertices and colors.

The matrix multiplication required for this project involved multiplying a large buffer of small (4x4) transformation matrices with one 4x4 scale matrix. Only one scale matrix was needed because the particles are all a uniform size. The results of these multiplications would then be written into OpenGL's matrix buffer. We used a cuda kernel to parallelize the multiplication and transfer of data. It works as depicted in the image below:



Matrix Multiplication

Transformation Matrices

Scale Matrix

OpenGL Matrices

In the image above, a thread is assigned to calculate the value of each number in the OpenGL matrix. For example, thread 0 (in red) would multiply the first row of the first transformation matrix with the first column of the scale matrix and write it into the first value in the first OpenGL matrix. At the same time another thread might be updating the last value of an OpenGL matrix by multiplying the last row of another matrix with the last column of the scale matrix. Rather than performing these multiplications in serial on the cpu, we were able to parallelize them with Cuda and write the results straight to OpenGL. Colors would be implemented in a similar manner. Each Cuda thread would be responsible for updating the 4 color values for each square.

One thing to note here is that the squares themselves don't move. The density (color) of each square simulates the fluid movement. Because of this and the fact that instanced rendering separates the memory buffers, we were actually able to calculate the transformation matrices once at the beginning of the simulation and then leave it for openGL to use later. This gives a performance boost for this project (less matrix multiplies) but also means that care would need to be taken to update the positions. For example, if the gpu can calculate the updated positions each frame it would be efficient but if it relies on user input it might require cpu to gpu data transfer.

Future work

Ideally, we would have benchmarked the performance of the different types of draw calls but we did not have time (and the rendering portion was not the focus of this project). We could potentially compare the serial implementation of the algorithm using instanced rendering vs the cuda interop instanced rendering vs cuda calculations copied to cpu copied to gpu, and possibly a serial algorithm with a serial draw call to show the performance results. The most likely result would be that the serial implementation outperforms cuda for small squares but falls behind with larger ones. The last two options are probably the least efficient as it would require data to be copied from the gpu to the cpu then back to the gpu each iteration or for thousands of draw calls to occur.

Other implementations we looked into on this interoperability with fluid simulations usually rely on using 2d textures and framebuffers to transfer the data. We decided on a different approach where we would use Cuda to update vertex buffers with information on color and positions. We believe this could also translate into a 3d implementation by combining our method with physics based rendering to produce a 3d fluid cube made of smaller cubes and using the density value to update the alpha channel of the cube (meaning squares with high density would be less transparent) but we did not have time to implement the project in 3d and test it. This would be valuable because 3d rendering of fluid is especially difficult and our approach would theoretically provide an efficient alternative to the texture implementation.

Conclusion

Overall, this algorithm posed some problems with parallelization, specifically with the lin_solve and set_bnd functions, but in the end was very parallelizable. Simulations based on this algorithm from Jos Stam are a mainstay in video game physics for things like water and smoke. Being able to parallelize this algorithm in ways like we accomplished or other ways is essential to the performance of these games. Also in scientific applications a simulation like this requires an extreme amount of particles to simulate real life. On top of that they require large time steps, which our algorithm being stable handles. Being parallelized allows for this increasing amount of particles in the simulation, as we proved in our performance analysis.

Source Code:

djperrone/StableFluids (github.com)