Real-Time Fluid Dynamics Optimization

Sean David Mcfarlane, University of Victoria,
Email: seanmcfarlane115@gmail.com,
Spring 2020

***Abstract*—Real time fluid dynamics simulations are a subset of fluid dynamics simulations that sacrifice accuracy for reduced computation cost. Performance is critical in real-time systems, which makes this simulation an ideal target for optimization. This report details three computation models that were utilized to optimize an existing sequential algorithm, and compares their performance. It was found that both the multicore SIMD solution and the CUDA solution significantly improved performance over the baseline on all resolutions and text environments, with the CUDA solution reaching over 73x speedup over the baseline when using an NVIDIA Tesla V100 GPU.**

I. Introduction

FLUID dynamics simulations have a wide variety of uses, from aerospace engineering to forest fire behaviour prediction. Real-time fluid dynamics simulations are a unique subset of fluid dynamics simulations that use a less accurate algorithm in exchange for real-time performance. The lack of accuracy to real life makes these simulations a poor choice for most applied use-cases, but they are ideal in situations when results only need to appear convincing, such as in CGI or video games applications. In real time systems, performance is absolutely critical, as it allows for shorter timesteps and higher resolutions. This makes these simulations a good candidate for optimization. The implementation used in this report is a modified version of the sample code accompanying the paper ”Real-Time Fluid Dynamics for Games” by Jos Stam [1]. The original implementation was designed to be as lightweight and barebones as possible, and is already fairly performant.

A. Algorithmic Description

This fluid dynamics simulation consists of two main layers: density and velocity. Both layers are represented by floats at discrete positions on an N*N sized grid. This simulation is in 2 dimensions only, but the algorithms used have been demonstrated to work in 3 as well. The density layer consists of a 2D matrix of floats, with each float representing the density of smoke at that cell on the grid. Densities are moved around based on the velocity at each cell and diffusion into neighbouring cells. The velocity layer consists of two 2D matrices of floats: one for x velocity, and one for y velocity. The velocity values are similarly diffused and self advected, which is to say, moved along the grid by their own velocities. The bulk of computations in this algorithm originate from solving the density and velocity values each timestep. This takes place in the functions dens_step() and vel_step, which are detailed below.

Firstly, the density solver (Figure 2): Here N is the resolution of the grid, x is the density grid for the current iteration,

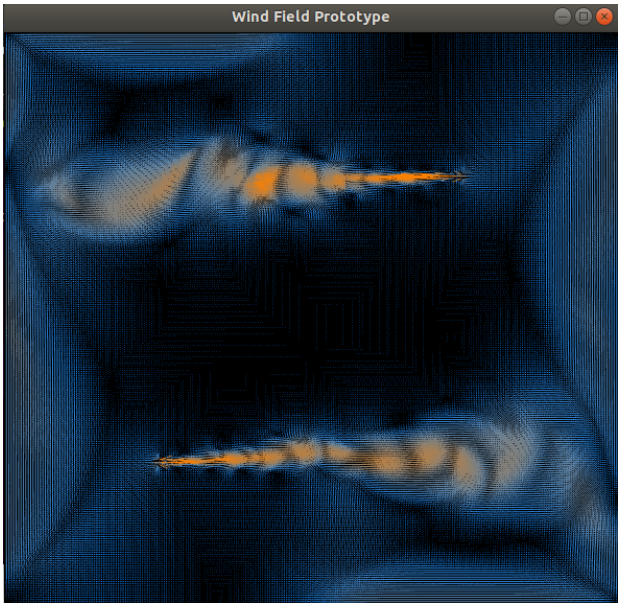


Fig. 1. Fluid dynamics simulation - velocity field simulation.

x0 is the density grid from the last iteration, u and v are x and y velocities, diff is the diffusion rate, and dt is delta time. The first step, add_source, adds new density to the grid from external sources like mouse clicks. The second is the diffusion step, where density at each cell is spread to neighbouring cells. This relates to how smoke/dust/dye disperses throughout a fluid, becoming more dilute. Gauss-Seidel relaxation is used for this step. The math behind the diffusion step is fairly simple but makes up the bulk of computation time due to running a 20 step loop each iteration. The second step, advection, ensures that densities are moved along the grid according to their velocity. Since the positions resulting from moving from grid cells along the velocity vector rarely land on cell centers, it becomes difficult and computationally expensive to move densities forward through the velocity field. Instead, this project implements backward tracing advection. With backward tracing, instead of considering where the densities are ”pushed” to each iteration, it determines what densities would be ”pulled” into each grid space by looking backward along the velocity vector. An interpolated average of nearby cells gives the new density for that cell.

Next is the velocity step, as seen in Figure 3. The first three steps of vel_step() are similar to the density solver, except each function is performed twice; once on the x velocities and once on the y velocities. The key difference is the addition of the project() function. The purpose of this function is to

```
void dens_step(uint32_t N, float *x, float *x0, float *u, float *v, float diff, float dt)
{
    base::add_source(N, x, x0, dt);
    SWAP(x0, x);
    base::diffuse(N, 0, x, x0, diff, dt);
    SWAP(x0, x);
    base::advect(N, 0, x, x0, u, v, dt);
}
```

Fig. 2. Density solver function.

```
void vel_step(uint32_t N, float *u, float *v, float *u0, float *v0, float visc, float dt)
{
    base::add_source(N, u, u0, dt);
    base::add_source(N, v, v0, dt);
    SWAP(u0, u);
    base::diffuse(N, 1, u, u0, visc, dt);
    SWAP(v0, v);
    base::diffuse(N, 2, v, v0, visc, dt);
    base::project(N, u, v, u0, v0);
    SWAP(u0, u);
    SWAP(v0, v);
    base::advect(N, 1, u, u0, u0, v0, dt);
    base::advect(N, 2, v, v0, u0, v0, dt);
    base::project(N, u, v, u0, v0);
}
```

Fig. 3. Velocity solver function.

ensure conservation of mass and prevent the simulation from becoming unstable. The project uses a technique called Hodge Decomposition to generate a mass conserving version of the velocity field, where incoming and outgoing flow to a cell must be equal.

B. Quality of Baseline

From the previous section it can be seen that the baseline is indeed difficult to accelerate without parallelism. The equations used and asymptotic complexity are minimal given what they seek to achieve. The memory efficiency of the baseline is not ideal. This was partially remedied in the optimized sequential implementation, which has a much lower CPI but higher instruction overhead. The original algorithm runs effectively on resolutions up to 512x512, which is enough precision for many use cases. The original paper by Jos Stam [1] provides a number of demonstrations of the variety of visual effects that can be created using the algorithm.

II. Optimized Models

A. Sequential Memory-Optimized

After identifying that the algorithm was extremely memory-bound, and the majority of execution time was taking place within one function, lin_solve(), the majority of optimization efforts were directed toward that function specifically. The first

```
#define IX(i,j) ((i)+(N+2)*(j))
```

Fig. 4. Original 2D Matrix to 1D array index function.

improvement to lin_solve() was to improve the spatial locality of data.

The original implementation uses a function called IX(i,j) (Figure 4) to map cells from a 2d grid to indices in a 1d array, in row-major order. All data associated with the cells on the grid is stored in this manner. Since lin_solve() accesses the horizontal and vertical neighbours of each cell, it was accessing at least 3 separate rows for every cell on the grid. To improve spatial locality, a new function was created that maps grid cells to memory in spatially-local chunks rather than rows, increasing the odds that all neighbours of a cell will be within 16 array indices of each other, regardless of matrix width. The new function, ZIX, which stands for Zoned IndeX, splits the matrix into square zones rather than rows (Figure 5). It takes a grid coordinate (i,j) and returns the corresponding array index it should be stored in. ZIX was effective in improving the spatial locality of data, but the loop in lin_solve() was still iterating over matrices in row form. This resulted in poor temporal locality, as nearby values in the same zone weren't being reused when they were already loaded.

```
static const uint32_t ZIX(const uint32_t x, const uint32_t y)
{
    const uint32_t zoneXc = (x >> divShift);
    const uint32_t zoneYc = (y >> divShift);
    const uint32_t zoneIndexc = zoneXc + zoneYc * zonesInRow;
    const uint32_t localXc = (x - (zoneXc << divShift));
    const uint32_t localYc = (y - (zoneYc << divShift));
    return (zoneIndexc * zoneSize) + localXc + (localYc * zoneLen);
}
```

Fig. 5. Optimized 2D Matrix to 1D array index function.

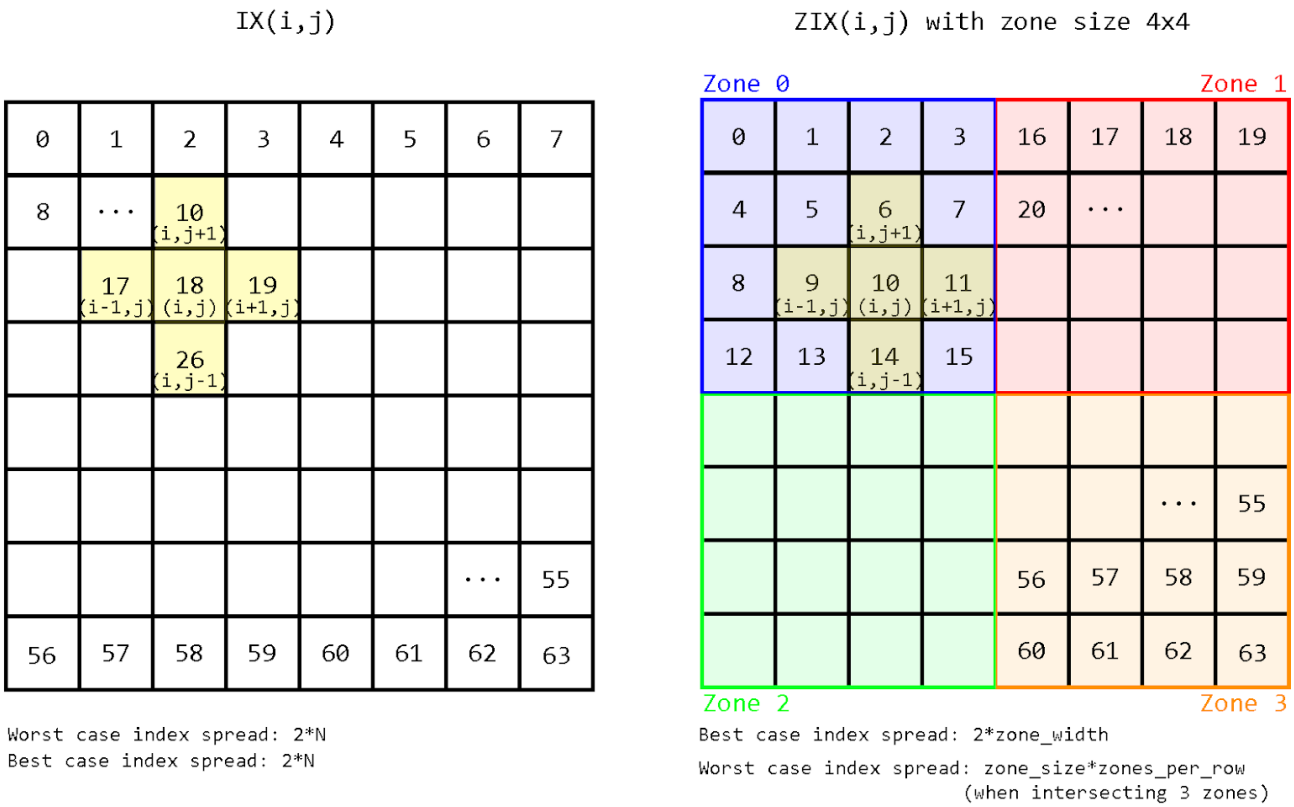


Fig. 6. Indexing of IX compared to ZIX. Numbers represent the array index assigned to each grid coordinate using either function. Yellow region indicates the cells accessed by lin_solve() when (i,j)=(3,3). Note the smaller spread of indices for ZIX.

This was remedied by restructuring the loop so that it iterates over the matrix in the same zoned order as the array. After splitting the matrices into subsections, V-Tune reports a substantial drop in cycles per instruction, indicating that the temporal and spatial locality of the 2d matrix access was the primary bottleneck. When all indices are in the same zone, the distance between the lowest and highest cell indexed is a constant 2*zone_width, regardless of overall grid resolution (See Figure 6). The constant distance will clearly perform better at scale when N is large. It's worth noting that the worst case distance of ZIX is actually worse than IX, but the average distance is much improved. IX only outperforms ZIX when one of the indexes accessed crosses into another row of zones above or below. Some consideration was put into selecting the zone size. The larger the zone size, the more operations can be performed within the borders of a single zone. But the larger the zone is, the more memory it requires. Beyond a certain cell expanding zone size results in poorer cache performance. After benchmarking zone sizes of 2^2, 4^2, 8^2, and 16^2, a zone size of 4*4 was found to be the most cache friendly for the resolutions used in this report. Since floats are 4 bytes long, each 16 cell zone takes up 64 bytes in memory.

B. Multicore SIMD

Unfortunately the optimizations made in previous solution were incompatible with a 4-wide SIMD configuration and had

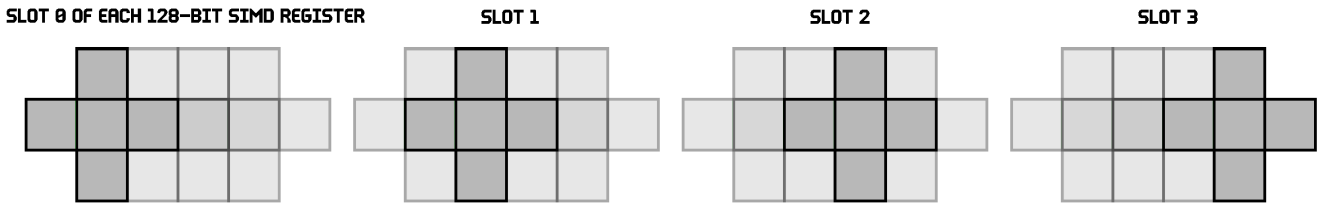


Fig. 7. Demonstration of how 4 iterations of the ”sliding window” of the lin_solve() function can be loaded into SIMD registers.

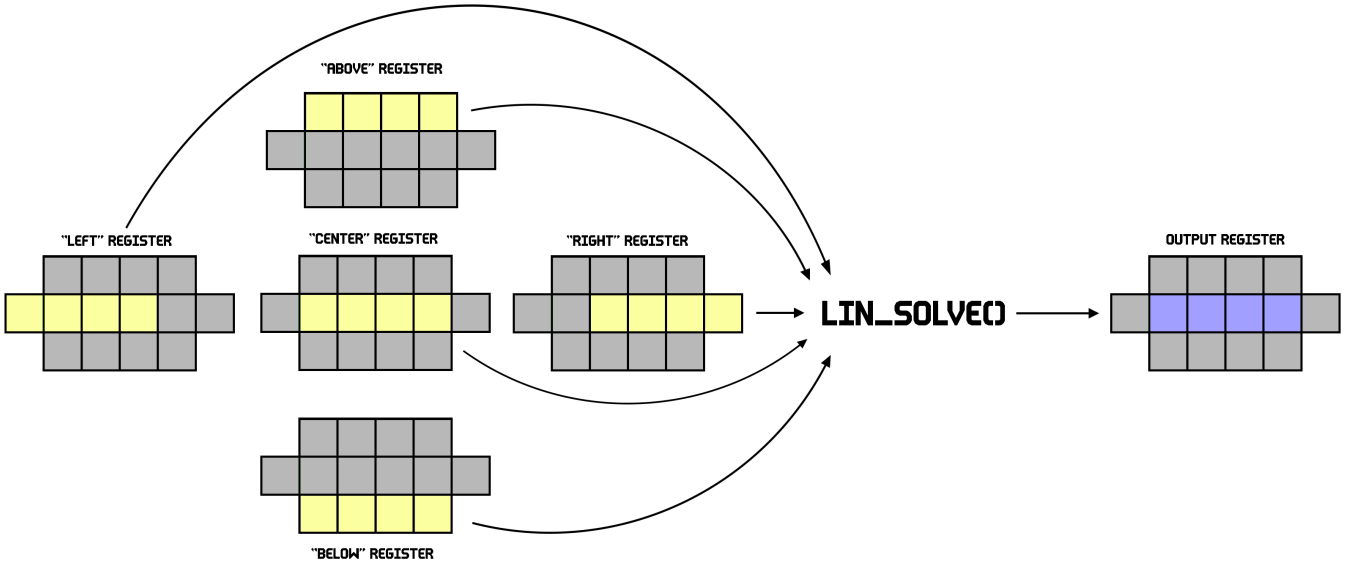


Fig. 8. The 6 registers used in each SIMD lin_solve() iteration, depicted by their relative positions to each other in the grid.

to be discarded for this solution. In order to ensure sequentially aligned memory accesses, the original row-major indexing scheme was used over zoned indexing. The performance benefits SIMD provides still more than make up for this compromise. The output for each grid cell is calculated using the values of the 4 neighbouring cells as input. This operation can be adapted to use SIMD instructions by calculating four horizontally sequential cells at once. Since the neighbouring cells are horizontally sequential as well, four SIMD ”neighbour” registers can be created.

The left and right registers are a bit more complicated to set up, since they are shifted 1 index out of alignment in each direction. This would normally make the use of SIMD impossible. The limitation can be circumvented, however, by use of the ”_mm_shuffle_ps” function. This function creates a new _m128 value that is a combination of two input _m128 values. By performing two shuffles in the illustrated configuration (See Figure 9), it’s possible to create left and right shifted registers. (Note: this means spatially shifted left and right, rather than bit shifted.)

The shuffling logic requires three _m128 registers in sequence to construct both the left and right-shifted, registers, but the ”current” and ”next” registers can be reused for the next iteration, so only one new register needs to be loaded for the middle row each iteration. As a result, only four registers

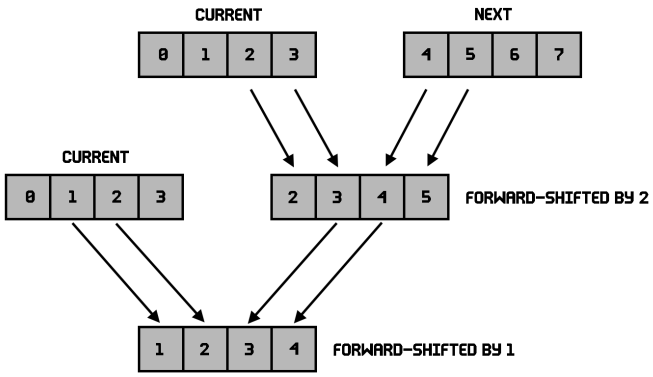


Fig. 9. Using two shuffle operations to create a 1-index offset in _m128 registers. Each shuffle must use only two values from each input, necessitating the second step.

need to be loaded per iteration, rather than six. As for race conditions, the output results are written to a different array, and the calculation for any grid cell is order-independent, preventing any potential issues.

The resulting output is identical to the sequential version, but cell values are calculated in batches of four. After completing the SIMD implementation, further speedups were

```
void lin_solve(uint32_t N, uint32_t b, float* x, float* x0, float a, float c) {
    float cInv = 1.0f / c;
    // Scalars are broadcast onto SIMD registers.
    __m128 _a = _mm_set_ps(a, a, a, a);
    __m128 _c = _mm_set_ps(cInv, cInv, cInv, cInv);
    uint32_t i, j, k;
    for (k = 0; k < 20; k++) {
        for (j = 4; j <= N + 4; j++) {
                __m128 mid_prev;
                __m128 mid_cur = _mm_load_ps(&(x[IX(0, j)]));
                __m128 mid_next = _mm_load_ps(&(x[IX(4, j)]));
                for (i = 4; i <= N + 4; i += 4) {
                        mid_prev = mid_cur;
                        mid_cur = mid_next;
                        mid_next = _mm_load_ps(&(x[IX(i + 4, j)]));
                        // Preparing the 5 SIMD registers
                        __m128 const center = _mm_load_ps(&(x0[IX(i, j)]));
                        __m128 const left = simd_lshift1(mid_prev, mid_cur);
                        __m128 const right = simd_rshift1(mid_cur, mid_next);
                        __m128 const up = _mm_load_ps(&(x[IX(i, j + 1)]));
                        __m128 const down = _mm_load_ps(&(x[IX(i, j - 1)]));
                        // Adding the four neighbouring cells together.
                        __m128 const adj1 = _mm_add_ps(up, down);
                        __m128 const adj2 = _mm_add_ps(left, right);
                        __m128 const adjacents = _mm_add_ps(adj1, adj2);
                        // Multiply by scalar and add to center
                        __m128 const scaledAdj = _mm_mul_ps(_a, adjacents);
                        __m128 const numerator = _mm_add_ps(center, scaledAdj);
                        // Multiply by scalar
                        __m128 const result = _mm_mul_ps(numerator, _c);
                        // Write result to output.
                        _mm_store_ps(&(x[IX(i, j)]), result);
                }
        }
        SIMD::set_bnd(N, b, x);
    }
}
```

Fig. 10. SIMD version of the lin_solve() function.

```
#pragma omp parallel for default(none) firstprivate(i, N, b, bnd, x0, x) schedule(static, 128)
```

Fig. 11. OpenMP compiler pragma used in multithreaded SIMD implementation.

attained by using multithreading with OpenMP. By adding the following pragma before the j-loop, the workload was split into 128-row segments and distributed to any number of available cores (Figure **??**).

Initially, since all cores were reading and writing to the same array, false sharing was a significant issue. This issue made memory efficiency poor enough to underperform the singlethreaded solution on all resolutions. The inclusion of the firstprivate statement allowed the arrays x0 and x to be hard-copied to each core, allowing them to be read without unnecessary overhead. This improved the multithreaded solution significantly, resulting in a similar runtime to plain SIMD on small resolutions, and over 200% speedup on larger resolutions.

C. CUDA

Previous implementations of this fluid dynamics simulation were focused on optimizing the lin_solve function, since it was responsible for the vast majority of execution time. Naturally, the CUDA implementation would seek to target this function as well. However, after replacing lin_solve with a CUDA equivalent function, tests reported that it was completing so quickly that the remaining sequential functions were now the bottlenecking factor. Work was then done to adapt the remaining simulation steps into CUDA kernels. In order to simplify the adaptation work, a macro was created that maps CUDA block/thread coordinates to grid cells. This allowed the sequential functions to be painlessly converted to CUDA kernels by replacing their FOR_EACH_CELL macros with

a new FOR_JOBS_IN_BOUNDS macro (see Figures 12&13 below). The FOR_JOBS_IN_BOUNDS macro uses a 2 dimensional block/thread layout to assign threads to grid cells. In order to test the ideal amount of grid cells computed by each thread, a global device variable ”d_CELLSPERTHREAD” was created. The macro divides work into block-width rows (Figure 14). These rows are then split among threads according to d_CELLSPERTHREAD. When d_CELLSPERTHREAD equals 1, every cell in the row is assigned its own thread. When d_CELLSPERTHREAD is equal to block-width, each thread will compute a full row of that block. This allows the user to tune thread load to find the optimum work distribution. In the tests conducted for this report, a block size of 16 and cells-per-thread of 1 was found to be optimum.

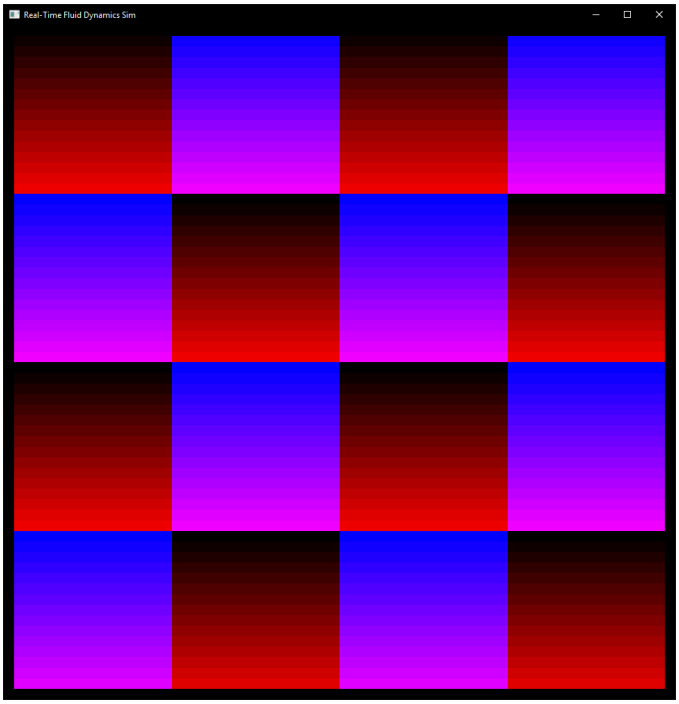


Fig. 14. Visualization of CUDA configuration on a 64x64 grid with CUDA block size 16. Each square represents one CUDA block, and each horizontal row represents a thread. Black pixels indicate excluded boundary cells.

This macro allowed most of the original baseline functions to be converted to CUDA functions with minimal effort. As seen below (Figures 15&16), the lin_solve function needed only to be split into a host and device function, with minimal modification to the interior content. The ”DIX” indexing function seen in the CUDA version is identical to IX of the original, but uses device variables instead.

After this groundwork was completed, it was trivial to convert each step to its CUDA equivalent. Profiling at this point showed that CUDA memory allocations and copies were taking up over 75% of execution time, so efforts were made to improve memory access. All device mallocs were moved to a single function that is called once during initialization, with memory frees likewise done a single time at the end of execution. Additionally, all simulation steps, including the diffuse, advect, and project steps of both the density solver and velocity solver function were integrated into one contiguous device-side sequence, allowing memory copies to and from the GPU to be done only a single time each iteration. This vastly improved memory performance. To further bolster memory efficiency, CPU side memory was allocated as pinned memory, rather than pageable, which allows cudaMemcpy to copy directly from CPU to GPU without using an intermediary array.

```
#define FOR_EACH_CELL for ( j=pad; j<N+pad; j++ ) { for ( i=pad ; i<N+pad; i++ ) {
```

Fig. 12. Original macro used to loop over all grid cells.

```
#define FOR_JOBS_IN_BOUNDS \
uint32_t i = (blockIdx.x*blockDim.x*d_CELLSPERTHREAD)+(threadIdx.x*d_CELLSPERTHREAD); \
uint32_t const j = blockIdx.y*blockDim.y+threadIdx.y; \
for (uint32_t jobnum = 0; jobnum < d_CELLSPERTHREAD; jobnum++) { \
if INBOUNDS(i, j)


#define END_JOBS i++; }
```

Fig. 13. CUDA macro to map threads to grid cells.

```
void lin_solve(uint32_t N, uint32_t b, float *x, float *x0, float a, float c)
{
    uint32_t i, j, k;

    for (k = 0; k < 20; k++)
    {
        FOR_EACH_CELL
            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)])) / c;
        END_FOR
        base::set_bnd(N, b, x);
    }
}
```

Fig. 15. Original macro used to loop over all grid cells.

```
__global__
void gpu_lin_solve(float* x, float* x0, const float a, const float c) {
    FOR_JOBS_IN_BOUNDS {
        DEVICE_CHECK_OOB
        x[DIX(i, j)] = (x0[DIX(i, j)] + a * (x[DIX(i - 1, j)] + x[DIX(i + 1, j)] + x[DIX(i, j - 1)] + x[DIX(i, j + 1)])) / c;
    } END_JOBS
}

__host__
void lin_solve(uint32_t b, float *d_x, float *d_x0, float a, float c){
    uint32_t k;
    for (k = 0; k < 20; k++)
    {
        CUDA::gpu_lin_solve<<<gridSize, threadCount>>>(d_x, d_x0, a, c);
        CUDA::gpu_set_bnd<<<bx, BSIZE>>>(b, d_x);
    }
}
```

Fig. 16. CUDA macro to map threads to grid cells.

III. Analysis and Testing

A. Setup & Reproduction

A link to the project repository can be found in the reference section at the end of the document [2]. Perf and VTune are effective for profiling the overall execution time and CPU performance of the project. nvprof has been tested to work on Linux for profiling GPU performance characteristics. An alternative headless executable, demo_perf.exe, was created for profiling, which skips the rendering code and runs a set number of iterations on only the simulation algorithm. The timestep is locked to a constant value of 1/60 to ensure that faster-running tests produce the same output. Compiling the render-mode executable requires the package libsfml-dev, but the headless executable has no additional requirements. For ease of testing, multiple implementations are included in the same project, and the desired implementation can be selected at runtime with a command line argument. This version of the project also allows two additional arguments to control the CUDA block size and number of cells computed per thread. Detailed usage instructions can be found in the README file accompanying the project. Rebuilding can be easily accomplished by running build.sh (Linux) or buildwin.bat (Windows) from the project directory.

B. Test Outline

The dataset consists of a 2 dimensional grid of velocities and a grid of densities. The tests were conducted primarily on 512x512 resolution, with further tests being conduction on 1024x1024 for the more performant models to demonstrate scalability. Two hard-coded sources of wind fill the wind field with turbulence. This serves as a replacement for user input in the headless version of the application, allowing tests to emulate realistic use conditions. Tests were conducted in three sets, each on different machines, detailed below:

Test Set 1:

- OS: Linux
- GPU: Unused
- CUDA Cores: N/A
- CPU: 4 cores, 2.5Ghz
- Profiler: VTune

Test set 2:

- OS: Windows
- GPU: NVIDIA GTX 1080
- CUDA Cores: 2560
- CPU: 4 cores, 3.4Ghz
- Profiler: VTune

Test Set 3:

- OS: Linux
- GPU: NVIDIA Tesla V100
- CUDA Cores: 5120
- CPU: 6 cores, 2.6Ghz
- Profiler: Perf

C. Test Results

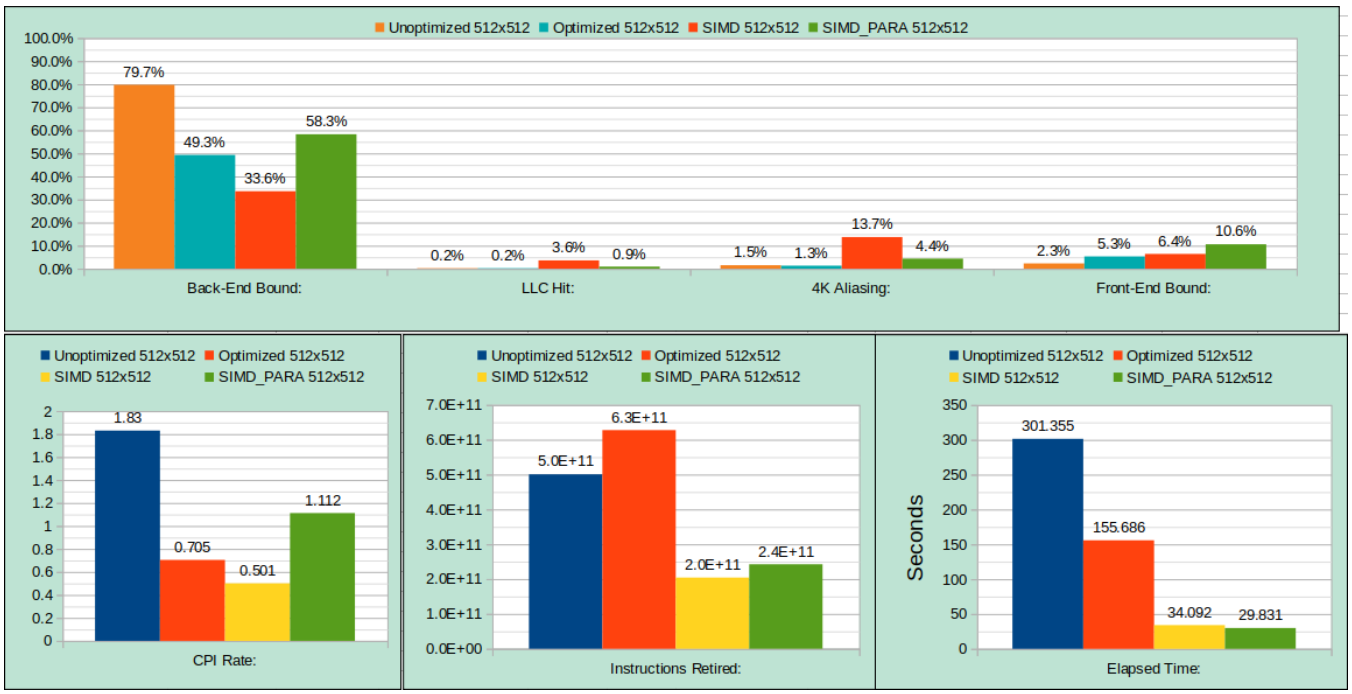


Fig. 17. Test set 1: VTune metrics at 512x512 resolution on Unoptimized, Sequential Memory-Optimized, SIMD, and Multicore SIMD solutions.

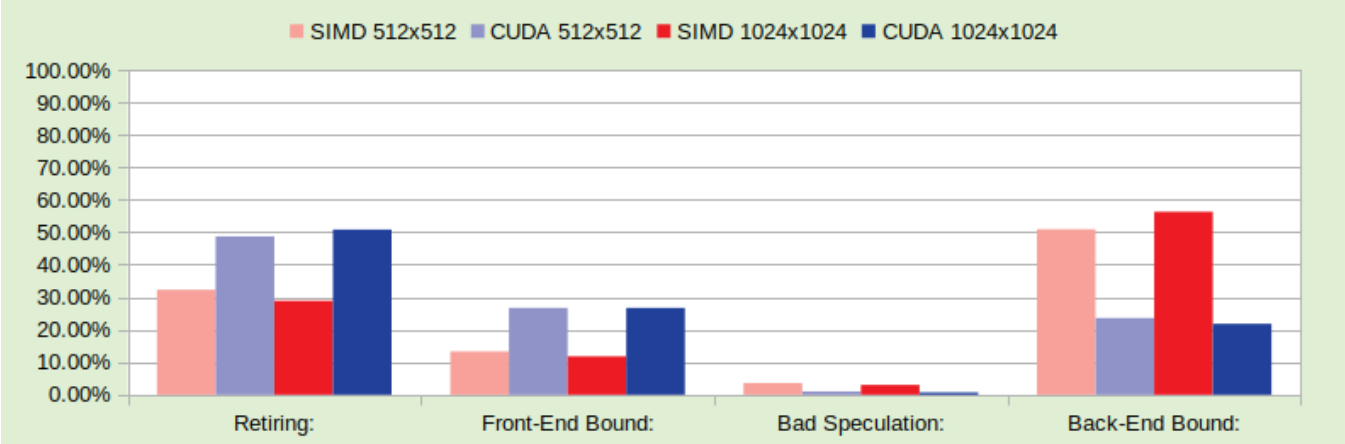


Fig. 18. Test set 2: VTune metrics at 512x512 and 1024x1024 resolution on Multicore SIMD and CUDA solutions.

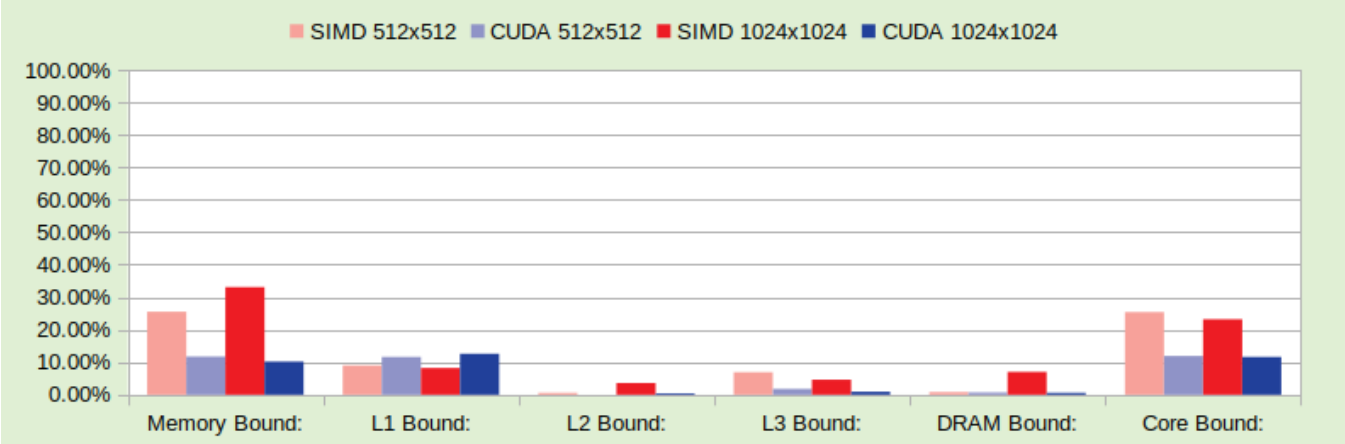
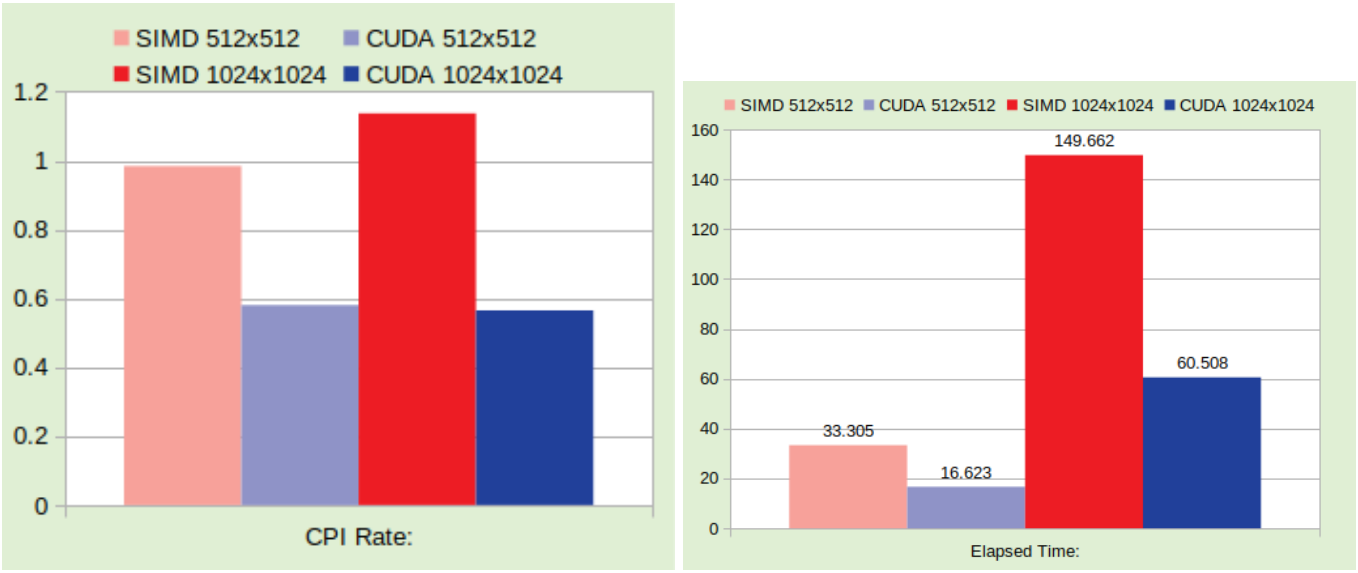


Fig. 19. Test set 2: VTune metrics at 512x512 and 1024x1024 resolution on Multicore SIMD and CUDA solutions.



(a) CPI rate

(b) Total execution time

Fig. 20. Test set 2: VTune metrics at 512x512 and 1024x1024 resolution on Multicore SIMD and CUDA solutions.

```
seanmcf@testbed:~/RTFD_2$ perf stat ./demo_perf.exe 1 0 512 500
Beginning test...
Test complete.

 Performance counter stats for './demo_perf.exe 1 0 512 500':

        137828.91 msec task-clock                #    1.000 CPUs utilized
              241      context-switches          #    0.002 K/sec
                0      cpu-migrations            #    0.000 K/sec
             1672      page-faults               #    0.012 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     137.835970668 seconds time elapsed

     137.800533000 seconds user
       0.028037000 seconds sys
```

Fig. 21. Test set 3: Perf metrics at 512x512 resolution on Unoptimized Baseline solution.

```
Performance counter stats for './Linux/demo_perf.exe 4 512 1000':

         77806.64 msec task-clock                #    3.990 CPUs utilized
            11207      context-switches          #    0.144 K/sec
               97      cpu-migrations            #    0.001 K/sec
             1704      page-faults               #    0.022 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      19.501238515 seconds time elapsed

      77.653252000 seconds user
       0.171472000 seconds sys
```

Fig. 22. Test set 3: Perf metrics at 512x512 resolution on Multicore SIMD solution.

```
Performance counter stats for './Linux/demo_perf.exe 5 512 1000 16 1':

          3625.14 msec task-clock                #    0.965 CPUs utilized
              116      context-switches          #    0.032 K/sec
                3      cpu-migrations            #    0.001 K/sec
             2951      page-faults               #    0.814 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       3.756841134 seconds time elapsed

       1.542744000 seconds user
       2.077079000 seconds sys
```

Fig. 23. Test set 3: Perf metrics at 512x512 resolution on CUDA solution.

```
Performance counter stats for './Linux/demo_perf.exe 4 1024 1000':

        240700.44 msec task-clock                #    3.105 CPUs utilized
            23874      context-switches          #    0.099 K/sec
              280      cpu-migrations            #    0.001 K/sec
             3245      page-faults               #    0.013 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      77.513870679 seconds time elapsed

     240.389667000 seconds user
       0.353691000 seconds sys
```

Fig. 24. Test set 3: Perf metrics at 1024x1024 resolution on Multicore SIMD solution.

```
Performance counter stats for './Linux/demo_perf.exe 5 1024 1000 16 1':

          8021.76 msec task-clock                #    0.983 CPUs utilized
               158      context-switches          #    0.020 K/sec
                 0      cpu-migrations            #    0.000 K/sec
              3483      page-faults               #    0.434 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

       8.160722191 seconds time elapsed

       4.591827000 seconds user
       3.425839000 seconds sys
```

Fig. 25. Test set 3: Perf metrics at 1024x1024 resolution on CUDA solution.

IV. Conclusions

Every previous method used in this project was severely limited by memory access speed. Despite all the optimizations made, the simulation was still primarily back-end bound. The CUDA implementation seeks to remedy this by leveraging the superior memory performance of GPUs, which are purpose built for computing highly parallel tasks, often on 2 and 3-dimensional data. The CUDA implementation moves the entire simulation loop onto the GPU, only copying back to the CPU to display the final result each iteration, and copying any input from CPU memory. It is clear from the included profiling results that offloading the compute-heavy work to the GPU significantly reduced back-end boundedness on the CPU. The CUDA implementation reduces accesses to lower level cache and avoids the false sharing pitfalls of the multicore implementations. By performing the entire iteration on the GPU, the simulation avoids unnecessary IO overhead from sending data across the PCI-E bus. Although support was included to have CUDA threads compute multiple cells per thread, testing indicated that using one thread per grid cell was optimal. The CUDA implementation performs significantly better on the 5120 CUDA core machine compared to the 2560 core machine. This would imply that a large portion of the workload is being effectively offloaded to the GPU in a scalable way. Overall performance of the CUDA implementation is superior on all resolutions.

References

[1] Stam, Jos. (2003). *Real-Time Fluid Dynamics for Games.* https://www.researchgate.net/publication/2560062_Real-Time_Fluid_Dynamics_for_Games

[2] CSC485C Project Repository - Real Time Fluid Dynamics Optimization https://github.com/SeanMcFarlane/RTFD