FAST FLUID SIMULATION - OPTIMIZING THE LATTICE BOLTZMANN METHOD

Johannes Gasser, Severin Klapproth, Douglas Orsini-Rosenberg, Rachel Schuchert

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We present an optimized implementation of the Lattice Boltzmann Method for three different boundary conditions: Lees-Edwards, Couette, and periodic. Our contribution lies in using basic C optimizations, well-thought-out locality optimizations, vectorizations, and some additional techniques to achieve a speedup of up to 90x. We focused exclusively on sequential optimizations, without considering parallelization. Our results demonstrate significant performance improvements, making our implementation suitable for a wide range of fluid dynamics simulations.

1. INTRODUCTION

Fluid dynamics simulations are crucial in fields like engineering and environmental science. Traditional methods, such as solving the Navier-Stokes equations, often face challenges with complexity and computational costs [1]. The Lattice Boltzmann Method (LBM) offers a more efficient and versatile alternative by solving the Boltzmann transport equation on a discrete lattice grid [2]. This paper focuses on optimizing the LBM to enhance its performance, aiming to make simulations more feasible and efficient for practical applications.

1.1 Motivation Real-time fluid dynamics simulations are essential for applications in various environments, from industrial processes to environmental modeling. The LBM is efficient but can benefit significantly from typical optimization techniques like strength reduction, loop ordering, and vectorization. Previous implementations, such as those by Philipp Mocz [3] and Callum Marshall [4], often lack these optimizations, limiting their performance. For instance, the 2D Python implementation by Mocz is specific and constrained, while Marshall's more versatile C++ version also does not fully exploit optimization opportunities. By addressing these gaps, we aim to improve the LBM's efficiency and applicability across different fluid dynamics problems, enhancing real-time simulation capabilities.

1.2 Contribution This paper presents optimized LBM implementations by adapting two existing versions into C. We built a comprehensive testing and timing infrastructure and applied various optimization strategies. Our work demonstrates significant performance improvements in LBM simulations with speedups up to 90x, providing a more efficient and flexible tool for real-time fluid dynamics simulations.

2. LATTICE BOLTZMANN METHOD

The Lattice Boltzmann Method (LBM) is a computational fluid dynamics technique that simulates fluid flow by solving the Boltzmann transport equation on a discrete lattice grid (Fig.1).

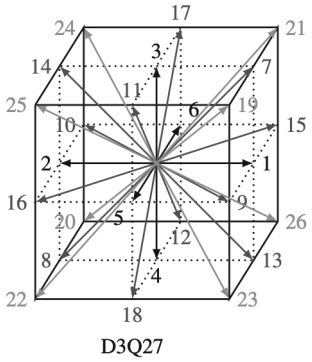

Fig. 1. D3Q27 lattice [2]

2.1 Algorithm. We start with an environment describing the initial state of the fluid, which is the particle distribution on a lattice, and specify the number of time steps. These particles move between lattice nodes and collide according to specific rules that conserve mass and momentum. This environment is represented by an $n_x \times n_y \times n_z \times q$ matrix, where n_i defines the lattice dimensions, and q is the number of discrete velocity directions. A common notation for lattices in the LBM is DdQq, where d represents the spatial dimension and q denotes the number of discrete velocity directions as can be seen in Fig. 1. The output is given by the fluid's new state after a given number of time steps, reflected in the updated particle distribution on the lattice. The main steps of the LBM algorithm are:

1. **Initialisation:** Define the lattice grid and initialize the

particle distribution functions based on the desired macroscopic properties, such as density and velocity.

2. **Streaming Step:** Particles move from their current nodes to neighboring nodes according to their velocity directions. This step propagates the distribution functions across the lattice.
3. **Collision Step:** At each node, particles interact (collide) and redistribute their velocities according to a collision operator, typically the Bhatnagar-Gross-Krook (BGK) approximation. This operator relaxes the distribution functions towards local equilibrium.
4. **Boundary Conditions:** Apply appropriate boundary conditions to handle the interactions of particles with the boundaries of the simulation domain.
5. **Macroscopic Quantities:** Calculate macroscopic properties like density, velocity, and pressure from the particle distribution functions at each node.

The LBM can be reduced down to three main steps: momentum, collision, and streaming. The momentum step involves calculating the properties, while the streaming step incorporates the boundary conditions.

2.2 Boundary Conditions. Boundary conditions are crucial in the LBM to accurately represent the physical constraints and interactions at the borders of the computational domain. Three common types of boundary conditions are the following:

- **Periodic boundary conditions** model spatially repetitive systems by allowing particles exiting one side of the simulation domain to re-enter from the opposite side with unchanged properties [2].
- **Couette boundary conditions** simulate flow between two parallel plates, where one plate moves at a constant velocity and the other is stationary, creating a linear velocity gradient. The fluid velocity matches the boundary velocities at the top and bottom boundaries due to the no-slip condition [2].
- **Lees Edwards boundary conditions** extend periodic boundary conditions to simulate shear flow. When particles cross the boundary, they are shifted by a velocity increment proportional to the shear rate, ensuring continuous shear flow across the domain [2].

2.3 Cost Analysis. When looking at the different steps of the LBM, the mix of operations differs greatly. While the streaming step mainly copies data around and therefore mostly calculates indices which are integer operations (iops), the collision step primarily performs floating point operations (flops). The momentum function is a mix of $\sim$40% iops and 60% flops. Therefore we decided to make separate cost analyses to better reflect each function's bottlenecks. For the streaming step, we decided to only consider iops. We consider arithmetic operations (addition, multiplication, subtraction, and division), comparisons, and the modulo operator as operations. We therefore end up with the following cost measure (with N_{op} the amount of appearances of an op operation):

$$C_{stream}(n_i, q) = (N_{iarith} + N_{mod} + N_{icmp})$$

For the collision step, we decided to only consider flops. There we consider addition, multiplication, and division operations. FMA operations are considered as an addition and a multiplication, so two operations for our cost analysis. We therefore end up with the following cost measure for the collision step:

$$C_{col}(n_i, q) = (N_{fadd} + N_{fmul} + N_{fdiv})$$

Lastly, for the momentum step, since there is a balanced mix of iops and flops, we decided to consider both in our cost measure.

$$\begin{aligned} C_{mom}(n_i, q) &= (N_{mom_i} + N_{mom_f}) \\ C_{mom_i}(n_i, q) &= (N_{iarith} + N_{mod} + N_{icmp}) \\ C_{mom_f}(n_i, q) &= (N_{fadd} + N_{fmul} + N_{fdiv}) \end{aligned}$$

With this cost measure, we can look at each function individually and reflect the true bottlenecks of each algorithm step.

3. IMPLEMENTATION DETAILS

For our initial starting point for this project, we adapted and converted two distinct LBM implementations into functional and tested C versions.

2D Baseline. The first implementation [3] is a high-level 2D Python implementation by Philipp Mocz from Princeton University. This implementation is a solution to a specific version of the D2Q9 LBM problem. It simulates fluid flow in a periodic box (where the fluid flows from left to right) where a cylindrical static obstacle is placed in the box and the fluid has to move around it. As the flow progresses, turbulence develops in the area behind the cylinder. The implementation uses periodic boundary conditions along the horizontal axis and the cylinder is modeled using a bounce-back boundary condition. The initial conditions assume uniform density and negligible external forces, focusing on the interaction between the fluid and the cylinder. The method is tailored for a very specific constrained problem within a controlled environment. Nevertheless, it serves a good purpose to get an initial understanding of how LBM works.

General Baseline. The second implementation [4], developed by Callum Marshall and available on GitHub, is

based on the time-step algorithm described in the book *The Lattice Boltzmann Method: Principles and Practice* [2] published by Springer. This C++ implementation is significantly more versatile compared to the first. Unlike the previous implementation, it does not include any static obstacles, thereby offering more flexibility in defining the simulation environment. Users can change the lattice setup, allowing it to work with various velocity sets, including D3Q15, D3Q27, and D2Q9. This implementation supports both 3D and 2D simulations, making it adaptable to a wide range of fluid dynamics problems. Additionally, it provides the user with the option to choose from three different boundary conditions: Lees-Edwards, Couette, and periodic boundary conditions. This flexibility in boundary conditions allows for the simulation of different types of fluid flows and interactions.

3.1 Limiting Project Scope. For both versions, we adapted the code to functional C versions and built a thorough timing and testing infrastructure based on the system used for the Code Expert Homework throughout this course.

When it came to optimizing, we decided to limit the scope of the project to only optimizing the more general version by C. Marshall. The reason for this decision is that the 2D Python implementation solves a very specific constrained problem, while the more general C++ version is more broadly applicable. Additionally, the optimizations are similar since they implement a similar scenario. Furthermore, the C++ version allows us to explore and compare the effects of different boundary conditions and their optimizations more thoroughly.

3.2 Describing the Environment.

System State. In our implementation the state of a Lattice Boltzmann system is represented by an LBM struct which saves the fixed parameters and the always-changing variables of our system. Parameters define our fluid's property, the dimension, and size of the considered system, the considered discrete velocity directions as well as their weighting used in the algorithm, and the information on what type of boundary condition will be considered. The current state of our system is represented in a handful of arrays which for our entire mesh define the density field (df), the components of the fluids velocity field(vf), the current particle distribution (cpd), and the previous particle distribution (ppd).

System Update. The dependencies of variables for updating the system state during a single time step are described with the following:

$$\begin{aligned} ppd &\leftarrow f_{collision}(cpd, vf, df) \\ (vf, df) &\leftarrow f_{momentum}(cpd) \\ cpd &\leftarrow f_{stream}(ppd, vf, df) \end{aligned}$$

We decided to implement these three steps as three separate functions, each taking as an argument the entire LBM struct and updating the system's variables accordingly.

3.3 Timing & Testing Infrastructure. We built a testing infrastructure around the three functions that on the one hand times and tests each function individually and on the other hand tests and times the whole algorithm. For testing the correctness of our implementations, we consider the baseline algorithm as ground truth and compare the whole system (particle distributions, density field, and velocity field) after every function execution to the values the baseline calculated. If a function produces correct results, we time it by measuring the amount of cycles it takes to execute the function. To do this we use the processor's Read Time-Stamp Counter (RDTSC). For consistent values, we first measure how many executions it takes to reach a minimum amount of cycles and then repeat the experiment with the calculated amount times 10. In the end, we calculate the arithmetic mean over all runs of a function.

4. OPTIMISATION

Before starting to optimize, we explored the distribution of the runtime across the three different steps of the algorithm. We concluded that all steps contributed significantly to the overall runtime of the algorithm which is why we decided to optimize them all simultaneously. Our optimization strategy was segmented into four distinct phases, each focusing on a specific aspect of the algorithm's performance and resource utilization. This approach allows us to assess which optimizations increased performance the most and also allows us to build an optimized version of the overall algorithm after each optimization phase. Throughout the various optimization phases, we monitored the contribution of each part of the simulation to the overall runtime to identify any bottlenecks that might require more attention. Our observations showed that the optimizations provided comparable speedups across all parts, and no significant bottlenecks developed. As a result, we continued to optimize all parts simultaneously.

4.1 Strength reduction, precomputing loop invariants, function inline expansion. In the initial phase of our optimizations, we focused on optimizing all functions of the simulation using basic C optimizations. This included simple strategies like function inline expansion, precomputations of loop invariant terms outside the inner loops, and applying strength reductions (e.g. replacing divisions with multiplications). We did not expect massive performance increases from this phase, as the optimizations are very basic and can most likely be performed by a compiler. However, performing this optimization step is still crucial to get a better theoretical understanding of the problem and bring our theoretically measured values like performed operations closer to reality.

4.2 Cache Optimisations. The next phase included rearranging instructions for the ideal cache locality. Reorder-

ing the nested loops, iterating over the discrete velocity directions in the outer loop and over grid points in the inner loop increases spatial locality for all sub-steps, and allows more computations to be done solely in the outer loop further reducing the operational intensity for streaming functions. We also tried introducing blocking to increase temporal locality. When updating the particle distribution in the collision function, the total amount of read data can be reduced by updating the particle distribution in place and swapping pointers of the two distribution arrays at the end, rather than loading both the previous and current distribution arrays.

4.3 memcpy, SSA, adapt memory layout. In the third phase, we tried to find a maximally optimized version of each sub-step, stopping short of manual vectorization. The main idea was to find implementations of the algorithm that gave the compiler the best chance to find an ideal code execution. Significant improvements were expected by changing the layout of how we saved the velocity arrays, splitting them up into three different data arrays, one for each velocity component. In all functions where these arrays were used, this would allow for a more efficient vectorization of the code by the compiler. A key optimization idea for the streaming steps was the heavy use of the `memcpy` function, i.e. by identifying groups of grid points that in memory are stored adjacent to each other and have identical streaming rules, allowing the copying of larger blocks of memory from the previous to the current distribution array, rather than iterating over each grid point and doing index calculation and value assignment individually. Additionally, it seemed meaningful to rewrite the many flops in the loop-body of the collision function in the static single-assignment coding style. Further manual loop unrolling was attempted, which did show varying effects for the different substeps.

4.4 AVX2. Our final optimization attempt was to vectorize floating point operations manually with AVX2 to see if we would be able to find a more efficient vectorization than the compiler. We decided against using AVX-512 instructions, as the architecture in our experimental setup described in section 5 does not support it.

4.5 Further Optimization Attempts. The attempt to fuse part of the already optimized sub-steps for further increase of cache locality did not result in any improvement of the performance, which is why we did not further pursue this.

5. EXPERIMENTAL SETUP

For our experimental setup, we used an AMD Ryzen 7 PRO 6850U with a clock frequency of 2.7 GHz and 8 physical cores. The processor uses the Zen3+ architecture. Its caching hierarchy consists of a 32 KB Layer 1 and a 512 KB Layer 2 cache per core and a shared 16 MB Layer 3 cache. Throughout the project, we experimented with various compiler flags to optimize performance. We ultimately decided to use gcc with the following flags: -O3 -mavx2 -mfma -funroll-loops -march=znver3. We chose these flags to give the compiler maximum optimization freedom, including the use of fused multiply-add (FMA) instructions, loop unrolling, AVX2 instructions, and specifying the microarchitecture for even more detailed optimizations. We compare the results of our manual optimizations to the results achieved through these compiler optimizations.

5.1 Implementation. For simplicity, we decided to test the system on cube-shaped grids of size n^3, meaning $n_x = n_y = n_z = n$ with n as the cube's side length. For n we used values 4, 8, 16, 32, 64, and 128. We also fixed the number of directions to 27, as in initial experiments, we did not observe different trends for different numbers of directions. We initialized the system with fixed, constant values for the particle distributions, velocities, densities, and all numerical constants appearing in the LBM. Again, choosing different values makes no difference as the calculations performed are the same. Every experiment was performed on its dedicated allocated data structure, to ensure we start with a cold cache.

5.2 Rooflines and Expected Bounds. To analyze our experimental results, we rely heavily on roofline plots. Roofline plots are very useful, as they clearly illustrate performance limits, highlight bottlenecks, and provide insight into the balance between computation and memory usage, helping us identify the most effective optimization steps.
For each of the functions we optimized, we created our own roofline plots, because, as noted in section 2, our functions have varying instruction mixes, so the achievable bounds vary from function to function. Each optimized version of a function will be represented by multiple same-colored connected crosses in the roofline plot, where each cross represents a different grid width. As grid widths, we used powers of 2 between 4 and 128. In almost all cases, bigger grids would perform worse, meaning as a rule of thumb we can say the crosses higher up in the plot represent the function run for smaller grid sizes. Further, we point out that up to a grid width of 32 all data appearing in the whole LBM algorithm should fit into the 16MB of the L3 cache. In each substep at least one of the arrays storing the particle distribution is accessed. One instance of the distribution alone will not fit into L1 cache starting from width 8, not fit into L2 at width 16, and not fit into L3 for width 64. Further for the width 128 the memory needed to store a single distribution array approached close to half a gigabyte. This is significantly larger than the L3 cache leading to all functions being entirely bound by the main memory bandwidth, and resulted in approximately identical performance for functions applied to even wider grids.
For memory bounds, we performed our own measurements

using a memory bandwidth benchmark[5]. We used the Sequential 256-bit copy metric, as this best reflects our experimental setup since every function reads data from one array, performs calculations, and writes the new data to another array. The 256-bit version makes the most sense, as we use AVX2 instructions and allow the compiler to do so as well. For the performance bound, we relied on the most optimistic instruction combination from Agner Fog's instruction manual.

6. EXPERIMENTAL RESULTS

After timing each of the functions we can present the following results from our optimizations.

6.1 Results Momentum function. As can be seen in Fig. 2, the baseline version seems to be in the compute-bound region of the roofline plot with some space for improvement, especially for larger grid sizes (lower data points). The first optimization phase did not improve the runtime considerably but moved the grid points to the bottom left. This is because we reduced the number of flops and iops with strength reductions and precomputations of values used throughout the loops. We conclude that these optimizations are similar to what the compiler already can do with the baseline version and our used compiler flags. In the second phase, the performance improved, especially for larger grid sizes. This can be seen since the points moved upwards and closer together vertically. This is because we made considerable improvements to the memory bandwidth by aligning the for loops to better consider the cache hierarchy. The third phase improved the performance even more. For this function, we performed manual loop unrolling which surprisingly performed better than the loop unrolling the compiler already does. Using AVX2 instructions in the fourth optimization phase as expected did not further improve performance. This is because the compiler already did well in translating the code from optimization phase three to AVX2 instructions.

To evaluate our achieved performance, we analyze the Momentum 4 function trying to find the expected compute bounds according to its instruction mix. It has two relevant loop structures. A nested for loop with $n^3q/4$ iterations and a single loop with $n^3/4$ iterations. The limiting operations in the first loop body are the four calls to `_mm256_storeu_pd()` which takes four cycles. The second is limited by the vectorized division and 3 multiplications taking 6 cycles per iteration. We therefore expect the operation performance to be bound at around

$$\frac{(7+2)n^3q + 4n}{\left(\frac{4n^3q}{4} + \frac{6}{4}n^3\right)} \text{ ops/cycle} \approx 8.7 \text{ ops/cycle}.$$

6.2 Results Collision function. The basic C optimizations did not lead to a significant decrease in runtime, the

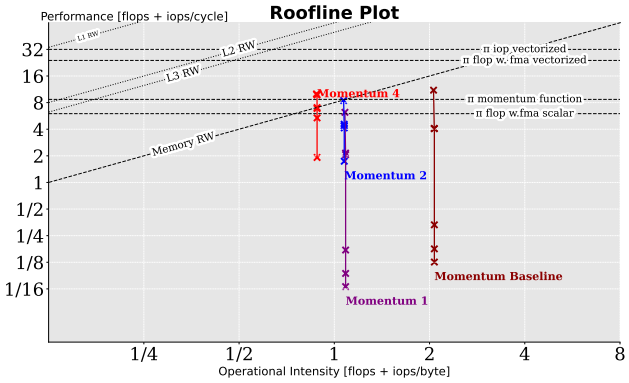

Fig. 2. Roofline - Momentum Function

minor speed-up that was observed mainly stemmed from replacing divisions with multiplications. The new theoretical count of performed operations gives us a roofline plot more representative of the code that is actually generated via present compiler optimizations (Collision 1 in Fig. 3). For the second version, the most significant improvement was gained by the increased spatial cache locality introduced by the reordering of the loop structure. Performing the computation in place of a single particle distribution array allowed for an increase in operational intensity without increasing the amount of performed operation, allowing higher performance for the function, especially when it is bounded by the bandwidth of main memory, which will be the case when collision is applied to larger grid sizes. The attempt to increase the temporal cache locality and performance by blocking for the iteration over grid points did increase performance when first tested on an Intel Kabylake processor. When tested on our fixed setup, the compiler already found an ideal way to avoid capacity misses, since the non-blocking code performed better on average than the version with blocking. The static single-assignment coding style as well as the new data layout of the velocity arrays helped the compiler find a better-performing execution and vectorization of the code, all the while manual loop unrolling seemed to worsen performance. The two effective changes made for the third optimization step led to reduced run time for grid sizes where the function was not yet bound by the main memory bandwidth. The graph for the third version was left out of the roofline plot, it would largely overlap with the graph for "Collision 4" since our manual vectorization only brought a minor improvement in performance. In the roofline plot, we can see nicely how the function performs consistently for some fixed compute-bound for small grid sizes, while for larger grid sizes it drops below the performance bound given through the main memory bandwidth.

As stated in section 2.3, due to the instruction mix, we only consider flops for the Collision function. In the most

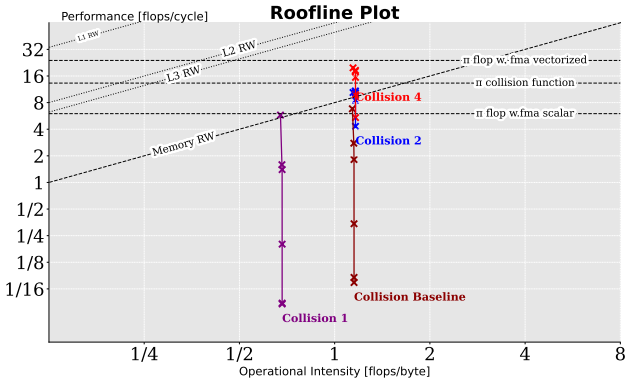

Fig. 3. Roofline - Collision Function

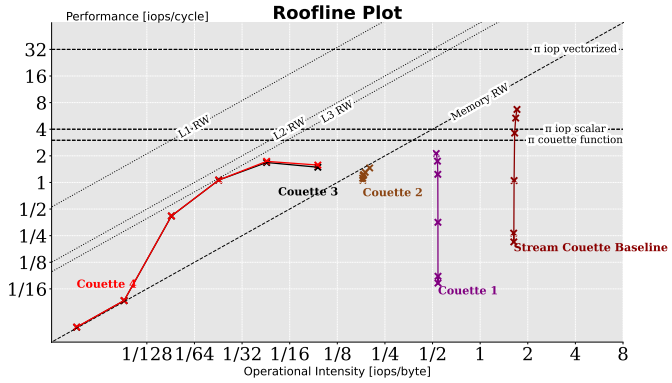

Fig. 4. Roofline - Couette Streaming Function

optimized version of the Collision function, there is a nested loop with $n^3q/4$ iterations, with $4 \cdot 20$ flops in each iteration, each of which takes 6 cycles per iteration. We therefore expect an upper bound on the performance to be around

$$\frac{20n^3q}{\frac{6}{4}n^3q} \text{ flops/cycle} \approx 13.3 \text{ flops/cycle}.$$

But we see the function for smaller grid sizes first being compute-bound above 16 flops per second and then dropping a good portion below the memory bandwidth for the larger grid sizes. The function outperforms our previously assumed ideal flop performance for smaller grid sizes likely because the two intermediate results — dot-product of the velocity and density field, as well as the velocity field norm — are independent of the direction iterator of the outer loop. Suppose enough memory is left in the cache during a run for storing these intermediate results. In that case, their recomputation can be avoided meaning only half of the counted flops would actually be performed by the CPU at each iteration. The increased memory traffic and near halving of the flop count greatly reduce operational intensity and lead to better runtime for small grid sizes. In cases where these intermediate results have to be stored in main memory, likely the cases that are already memory bound, the higher memory traffic leads to the performance worsening. A final processor-specific optimization could be introducing a case distinction, deciding whether the pre-computation and storing of the $2n^3$ intermediate values improves performance or not.

6.3 Results Streaming function. All streaming functions roughly behaved the same after the optimizations we performed. Therefore, we will focus on one of the three functions but every statement we make for this function also applies to the other two versions of the streaming step.

As can be seen in Fig. 4, the baseline lies in the compute-bound region of the roofline plot when considering the memory bandwidth of L1, L2, or L3 caches. The first optimization did not improve the performance at all but moved the data points to the bottom left to the memory-bound region of the roofline. This is because now our code reduced the number of iops and better reflects the optimized code the compiler produces. In optimization phase 2, the cache performance improved drastically, as we changed the loop order to better reflect the memory alignment of the data structures. After this phase, the grid points move even further towards being memory-bound. In phase 3, we got rid of most of the index calculations by using a `memcpy` approach that uses information about in what direction we stream to copy whole chunks of our particle distributions array instead of deciding for each point where it should move. After phase 3, we can say that this function is generally memory-bound except for the smallest grid size. This initial compute bound can be explained because, for small grid sizes, the floating point operations performed at the boundaries of the y-axis will impact the execution more than for larger grids. We can also see, that after a grid size of 64, we are fully bound by the memory bandwidth and reach comparable results to the memory benchmark that we performed on our system. Using AVX2 for the streaming step gave little to no improvement, especially for larger grid sizes. This is on the one hand because the compiler already efficiently vectorized large parts of the code and on the other hand because floating point operations, on which we focused for vectorization, make up a very small part of the streaming step in general. Iops were quite hard to vectorize for this problem, as the `memcpy` approach used a convoluted structure of if-else statements where vectorization would have little effect. Iops are mainly compromised of instructions having throughput 4 or 1 and lower. As an optimistic estimate for the upper bound, we can say the relation of these two types of instructions is 2:1 leading to an assumed maximum for performance of 3 integer operations per cycle.

6.4 Performance. In Fig. 5 we show the performance of the LBM simulation comparing all boundary conditions from the initial version to our optimized version. For small

grid sizes, the performance improvement is minor because the data fits in the L1 cache, so the improved code locality doesn't affect the runtime much. The optimized versions perform best at grid size 8^3, but then the performance drops as operational intensity decreases for streaming functions. As the grid size increases, the computations become limited by memory bandwidth. For larger grid sizes beyond the initial small ones, our optimized versions show performance increases by a factor of 4 to 25 times. For this performance plot, since it covers the full simulation, we used a cost measure where we considered both iops and flops to calculate performance, i.e.: $C_{LBM} = C_{col} + C_{mom} + C_{stream}$

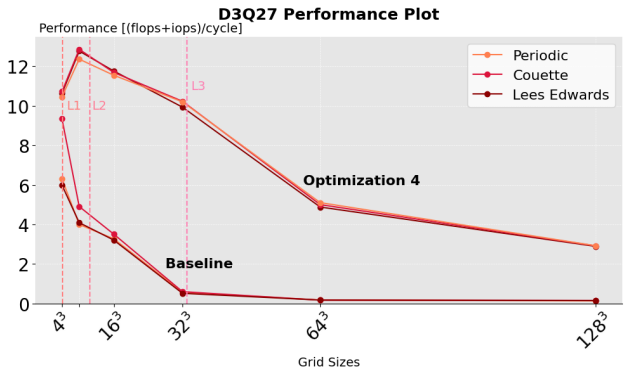

Fig. 5. **Performance Plot:** LBM simulation using different boundary conditions for the baseline and the final version.

6.5 Runtime & Speedup. The runtime plot in Fig. 6 illustrates the impact of each optimization step on the LBM simulation using the Lees Edwards boundary condition. It shows consistent runtime reductions across all tested grid sizes due to the optimizations. The most significant improvement is observed with the second series of optimizations, which effectively reduces cache misses across different memory levels. This leads to a substantial reduction in cycles for all grid sizes. In contrast, the other optimization steps primarily show an effect when the grid size causes the simulation to be compute-bound, and only cause marginal improvements for memory-bound grid sizes.

Table 1 presents the achieved speedups for various boundary conditions and grid sizes. The data highlights significant speedups, particularly at larger grid sizes, with the highest speedup reaching 90.9x for the LBM simulation using the Couette boundary condition at a grid size of 64^3. These results underscore the effectiveness of the optimizations in decreasing the execution time of the LBM simulation across different configurations and boundary conditions.

7. CONCLUSIONS

Our work demonstrates the potential for substantial performance gains through careful optimization. Our optimizations have significantly reduced operational intensity and improved performance across all grid sizes. When compared to the original implementation by Callum Marshall [4] we demonstrated speedups of up to a factor of 90. Our final function implementations seem very close to their expected compute or memory bounds for most input sizes, indicating that there's not much room left for further optimization in a sequential manner. A sequential run is unlikely to be able to compete with most parallel implementations since LBM's design allows for very efficient parallel executions. It would be interesting to see if key insights gained when optimizing a sequential setting might offer significant help in the development and optimization of an efficient parallel implementation. Even for a relatively small model (128^3), the run time for a few amount of timesteps already seemed excessively long using the baseline implementation. With the LBM being so widely used, a purely sequential but efficient implementation might make the method more accessible to interested parties with no access to or understanding of massively parallel computation frameworks. An implementation like ours would allow these much faster executions of LBM models, making it feasible to simulate more complex scenarios or larger environments within a reasonable timeframe.

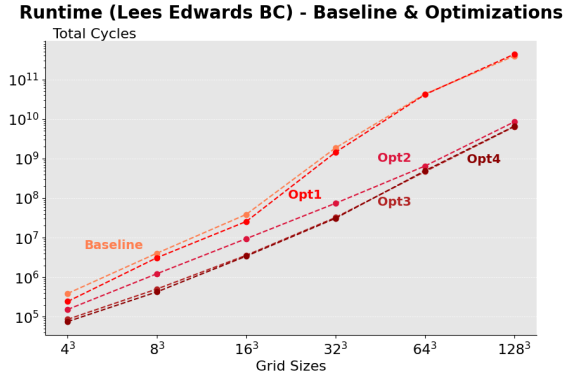

Fig. 6. **Runtime Plot:** LBM simulation using Lees-Edwards bound. cond. for different versions of our code

Table 1. Achieved Speedup

N	Periodic	Couette	Lees E.
4	4.7	3.6	4.8
8	9.6	7.8	9.4
16	11.6	10.2	11.5
32	41.6	55.8	60.5
64	88.3	90.9	83.3
128	69.8	62.3	59.4

8. CONTRIBUTIONS OF TEAM MEMBERS

Johannes. Translated the 3D C++ version to C, in which I translated the C++ object structure to multiple arrays passed between functions. Later took Douglas' timing infrastructure he built for the 2D version of our problem and adapted it to the 3D version. Added an infrastructure similar to the one we worked on Code-expert in which we can dynamically add functions we want to include in our testing and timing and split the timing of the whole function to the timing of the different sub-steps. Later mainly worked on all four optimisation steps of the Lees-Edwards streaming step. Found a `memcpy` approach to the streaming steps, which was then implemented for the periodic and the Lees-Edwards streaming approach. Lastly supported Rachel in writing an AVX version of the momentum function.

Severin. Rewrote Johannes initial 3D implementation from a version with multiple function arguments to a struct-based implementation. Implemented CSV file output in the main function of timing.cpp, which saves needed performance data. Wrote an initial version of Python code used for roofline plotting. Adapted main function from timing.cpp to take command line arguments allowing to specify grid dimensions and which parts should be tested at runtime. All work that was done for optimising the collision function. Attempt to fuse two sub steps for further optimisation.

Douglas. In collaboration with Rachel I built the infrastructure for the 2D Python implementation which was discontinued. Later on focused on optimizing the Couette Streaming Function. Performed basic C optimizations like strength reductions, made use of better locality by better structuring the loop and making precomputations. Relied on `memcpy` to get rid of some loops that copy single array elements. In a final optimzation I vectorized the most optimized couette boundary condition function using AVX2.

Rachel. Collaborated with Douglas on implementing the 2D LBM Python version into C. Contributed to the boundary condition comparison to define our objectives. Optimized the periodic and momentum functions through multiple optimization steps, including strength reduction, index pre-calculations, loop reordering, and unrolling. Worked with Johannes to implement AVX instructions for the momentum function. Created Jupyter notebooks for generating various plots, including performance, runtime, bottleneck, and roofline plots. Adapted Severin's Python roofline code for our purposes.

9. REFERENCES

[1] Cyrus K. Aidun and Jonathan R. Clausen, "Lattice boltzmann method for complex flow simulations," *Annual Review of Fluid Mechanics*, vol. 42, no. 1, pp. 439–472, 2010, Lattice Boltzmann Method for Complex Flow Simulations.

[2] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen, *The Lattice Boltzmann Method: Principles and Practice*, Graduate Texts in Physics. Springer, 1st edition, 2017.

[3] Philip Mocz, "Create your own lattice boltzmann simulation (with python)," Princeton University, 2020, https://medium.com/swlh/create-your-own-lattice-boltzmann-simulation-with-python-8759e8b53b1c.

[4] Callum Marshall, "Lbm," Github, 2019, https://github.com/callummarshall9/LBM.

[5] Zack Smith, "Bandwidth: a memory bandwidth benchmark," zsmith.co, 2012, https://zsmith.co/bandwidth.php.