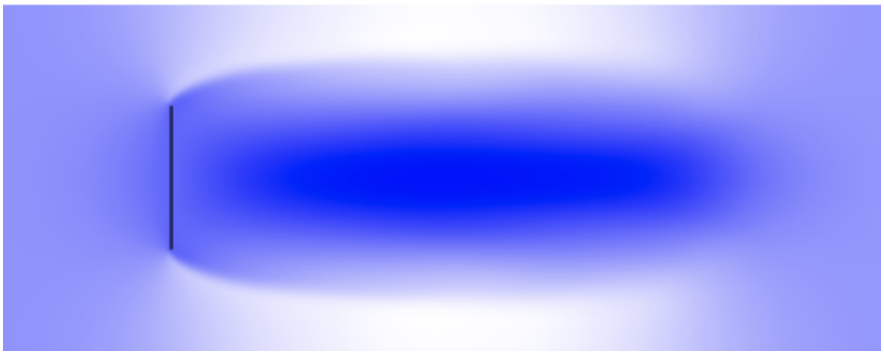
2D Fluid Dynamics with the Lattice Boltzmann method

Peter-Jan Gootzen (13837478) Ruben Stap (11269146)

April 2021



Contents

1 Introduction

This report describes the optimization of a C-based 2D fluid dynamics simulation. The Lattice Boltzmann method is used in this simulation. This method can be used to simulate e.g. arterial flow and particle suspensions of which the latter often occurs in cosmetics [1]. Note that there are many different fluid simulation methods each with their own strengths and weaknesses [3]. This implies that the Lattice Boltzmann method is not always a suitable choice. Apart from the mentioned applications, the method has also been used in other fields such as quantum mechanics and image processing [3].

The starting point of this project was the course material from the *Physics 3300* course at *Weber State University*, Utah U.S.A[4]. This course provides a introduction to the Lattice Boltzmann method and a Python reference implementation. Our initial (sequential) reference implementation is a quite literal one-to-one port of this Python reference implementation.

2 Lattice Boltzmann Algorithm

In this section the Lattice Boltzmann algorithm will be explained using the course from Weber as reference material[4]. The reader should be alert of possible mistakes due to the fact that the authors are not experts in physics.

Given the state of a fluid at time t, this algorithm determines the state of the fluid at $t + \Delta t$. A fluid is represented as a collection of molecules which can move across a lattice. Before discussing how the algorithm calculates a state update some prerequisites will be discussed.

2.1 The DS2Q9 lattice

The choice was made to use a so-called D2Q9 lattice. This lattice type is visualized in Figure 1. As can be seen in this figure, this lattice can only be used to represent a 2D fluid. Molecules can reside in the different square lattice cells each of which have area $A = (\Delta x)^2$. Movement of these molecules between the cells is restricted to the nine vectors $\vec{e_i}$. Note that this movement is in terms of a 2D cell index. The possible displacement vectors $\vec{d_i}$ and velocity vectors $\vec{v_i}$ are defined as $\vec{e_i}\Delta x$ and $\vec{v_i}(\Delta x/\Delta t)$ respectively.

2.2 Probability of $\mathbf{v_i}$

$$\mathbb{P}(\vec{u}, \vec{v_i}) = w_i \left(1 + \frac{3\vec{e_i} \cdot \vec{u}}{c} + \frac{9}{2}\left(\frac{\vec{e_i} \cdot \vec{u}}{c} - \frac{3}{2}\frac{\vec{u} \cdot \vec{u}}{c^2}\right)\right) \quad (1)$$

$$\text{where } w_0 = \frac{4}{9}, w_1 = w_2 = w_3 = w_4 = \frac{1}{9}, w_5 = w_6 = w_7 = w_8 = \frac{1}{36} \quad (2)$$

Equation 1 is a critical component of this algorithm. If the molecules within a lattice cell have a macroscopic velocity $\vec{u}$ this equation indicates the probability

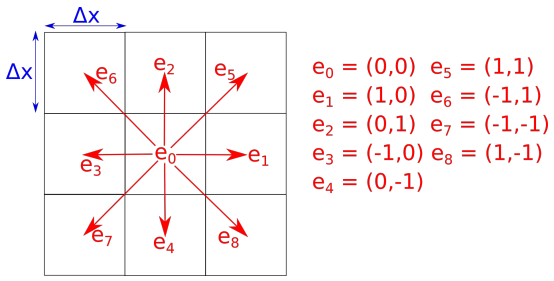


Figure 1: Visualization of the DSQ9 grid.

that these molecules will have velocity v_i after reaching thermal equilibrium. The variable c is defined as $\Delta x/\Delta t$ which is the constant speed with which molecules move across between lattice cells.

2.3 A state update

Assume that there is a fluid state for which we want to determine the next state. A fluid state consists of 9 directional number densities n_i for each lattice cell. Each directional number density n_i can be seen as the ratio N_i/A where N_i is the amount of molecules with velocity v_i. Figure 5 visualizes the number densities that are saved for a lattice cell.

2.3.1 Stream

In the first step of a state update all number densities of each cell are moved to other cells, thus streaming the lattice. This is movement visualized in figure 2 for the middle cell. For example n_2, which correspond to movement in the north direction, is moved from the middle cell to the top cell. The number density n_0 stays in the middle cell as it corresponds to no movement.

2.3.2 Bounceback

In this simulation there can be boundaries in the fluid. If molecules hit these boundaries they should bounce back, i.e. have their velocity reversed $(\cdot -1)$. In the previous step it could be the case that molecules moved inside boundaries, therefore in step two a bounceback is performed. To model the "bounceback" all number densities within boundaries are moved in the opposite direction. Figure 4 displays a column boundary and a single fluid cell on the left. In the previous step, n_5, n_1 and n_8 are moved within the boundary. These correspond to the northeast, east and southeast number densities. In the final step these number densities are moved to the northwest, west, southwest number densities.

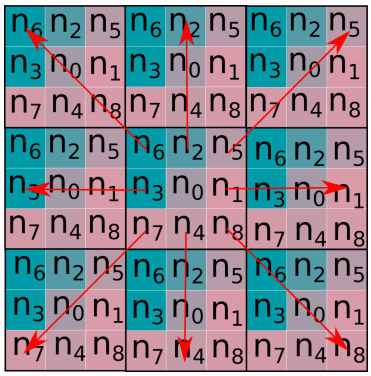


Figure 2: Visualization of the movement of number densities of the middle cell in the last step of a state update.

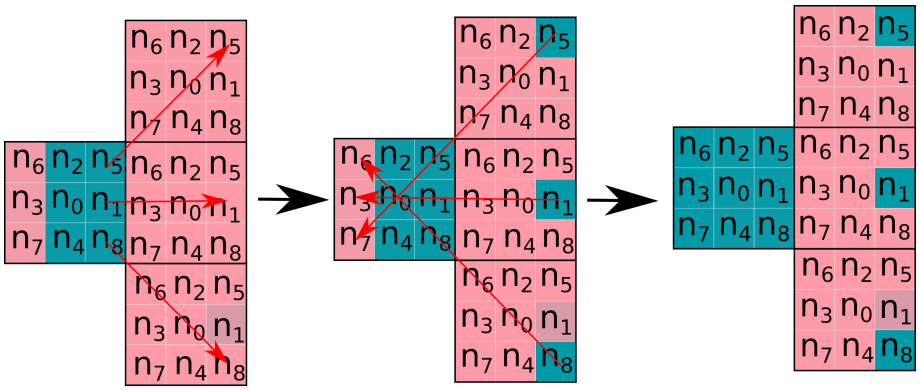


Figure 3: The leftmost figure indicates the action corresponding to the first step. The middle figure indicates the fluid state as a result and the action corresponding to the second "bounceback" step. The rightmost figure displays the result of this step.

Figure 4: Left: stream into object grid points. Middle: the molecules are "bounced back". Right: result of the bounceback step.

2.3.3 Collide

The following four steps are together called the collide part of the Lattice Boltzmann algorithm.

The third step in the algorithm is to determine the total number density ρ for each cell. For each cell this is the sum of each "directional" number density.

The fourth step of the algorithm is to determine the velocity of each cell. The horizontal component u_x and the vertical component u_y are determined according to equations 3 and 4. Both values are restricted to the range $[-1, 1]$.

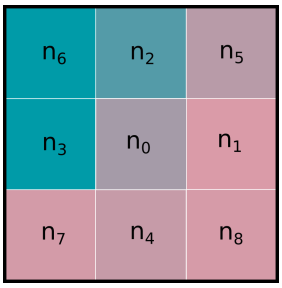


Figure 5: Visualization of the 9 number densities that are saved for each lattice cell. A green/blue background color indicates a low value while a red background value indicates a high value.

Figure 5 visualizes the number densities that are saved for a lattice cell. The value of u_x is positive if the majority of the density in n_6, n_3, n_7, n_5, n_1 and n_8 is located on the right. The value of u_y is also positive if the majority of the total density in n_6, n_2, n_5, n_7, n_4 and n_8 is located on the top. Both quantities are negative in the inverse case. In the figure $u_x > 0$ and $u_y < 0$.

$$u_x = (n_5 + n_1 + n_8 - n_6 - n_3 - n_7)/\rho \tag{3}$$
$$u_y = (n_6 + n_2 + n_5 - n_7 - n_4 - n_8)/\rho \tag{4}$$

In the fifth step the algorithm determines, for each cell, the 9 number densities when thermal equilibrium has been reached. For each cell a value of n_i^{eq} can be determined by calculating $\mathbb{P}(\vec{u}, \vec{v_i})\rho$. This expression calculates the expected number density of molecules that have velocity v_i after reaching thermal equilibrium. Because there might not be a thermal equilibrium in a cell due to the fact that the molecules in this cell have "just arrived" in the streaming step.

The sixth step of the algorithm is to update the 9 number densities of each cell based on equation 5. In this equation the constant $\omega \in [0, 2]$ and is inversely related to the viscosity of the fluid. The viscosity can be seen as the thickness of a fluid.

$$n_i^{new} = (1 - \omega)n_i^{old} + \omega n_i^{eq} \tag{5}$$

2.3.4 Steady flow

The seventh and last step is to reintroduce a flow into the lattice with a certain configuration. This is to make sure that the steady flow that one wants to simulate is an invariant. If this step were to be omitted, the fluid would eventually come to a stand-still.

A good real-life example of this is a wind tunnel or a water stream. This step simply inserts certain values of density into the lattice at given positions for certain directions, for example on the left side of the lattice with higher densities in the east sided directions than in the west sided directions, thus creating a

rightward flow. The exact number densities which are to be inserted can be generated by setting the desired $\vec{u}$ and ρ and then determening the values of $n_i^{eq} = \mathbb{P}(\vec{u}, v_i)\rho$.

3 *C* reference implementation

As previously mentioned, the reference implementation is a quite literal port of the Python implementation from the reference material[4] to C. This implementation consists of four kernels: *stream*, *bounceback*, *collide* and *steady flow*, that correspond to the steps in 2.3. The program is ran for a certain amount of iterations in which all four kernels are invoked. For each grid point all those invocations correspond to a state update as has been described in Section 2.3. These four kernels will be further expanded upon in the analytical model of the reference implementation in Section 5.1. The data layout and our validation procedure will be further expanded upon in the subsequent sections.

3.1 Data layout

As explained in Section 2.1, the algorithm uses a 2D grid where each grid point has 9 separate values representing the molecule density. In the reference implementation this is laid in memory as: `float (*grid)[9][H][W]`, this being a row-major 3-dimensional array where the first axis represents the direction, the second axis (with length H) represents the y-coordinate and the third and last axis (with length W) represents the x-coordinate. In this report the term *grid size* will be used to refer to $H \times W$, as this is the configurable part of the effective grid size.

The second piece of interest is the way objects are encoded in memory. Any grid point (y, x) can either contain an object or not. Therefore this can be simply laid out in memory as follows: `bool (*object_grid)[H][W]`, this being a row-major two-dimensional array of size $H \times W$ where each boolean represents whether there is an object on the corresponding (y, x).

3.2 Validation

The reference implementation and all subsequent implementations are verified to ensure that no changes inadvertently altered the correctness of the implementation. Initially it was planned to use the Python implementation as the baseline "correct" implementation to compare against. However while porting the bounceback phase of the algorithm a mistake was noticed in the Python implementation. Having confidently concluded that this was the only difference in the output of both implementations, we opted to fixing this mistake in our implementation and taking our implementation as the "correct" baseline point for validation.

Verification of an implementation is done by comparing the minimum, maximum and average of each number density array and velocity arrays to the

corresponding values of the C reference implementation after every iteration. As all these values are floating-point numbers, the precision of the data type is taken into a account and only differences that are not caused by floating-point imprecision are seen as incorrect.

4 Research question & experimental setup

To performance engineer an application, a research question and goal has to be defined to form the base for the experimental setup, models and optimizations. During this project a single time-step of the simulation will be optimized. Hoping to achieve (atleast) real-time fluid dynamics simulation for high-resolution grid sizes (i.e. $\geq$ 1080p). Leading to the following formulation of the research question:

> How fast can a single time-step in the 2D fluid dynamics simulation be performed?

Our experimental setup consists of the various parameters that are given to the implementations and the various performance metrics that are optimized for and evaluated upon. All experimentation, analysis and benchmarking is performed with the DAS-5 using the Intel Haswell microarchitecture.

4.1 Inputs & parameters

Inspecting the reference implementation, the following inputs can be defined:

1. the size of the density matrix, defined as $H \times W$
2. the initial values in the $H \times W$ density matrix
3. grid points on which object boundaries lie (i.e. amount and location)
4. the *steady flow* configuration, i.e. where and how much density is spawned in every iteration

These inputs are abstracted away from the reference implementation described in 3, as their definition will have to stay consistent across all implementations that might alter data layouts.

Not all these inputs will be defined as parameters to our experimental setup however. The second input, which does not affect performance in any way will not be used as a parameter. Because it only alters the values **in** the calculations and data movements, not the amount of computations, making it a poor choice for a parameter. The fourth input (steady flow configuration) does affect performance as it must include a certain amount of stores to the density matrix. However it would not be an interesting parameter to analyze as it is quite simplistic, so this input will not be included as a parameter to limit scope. Therefore we will fix the initial values and fix the steady flow configuration to a rightward steady flow.

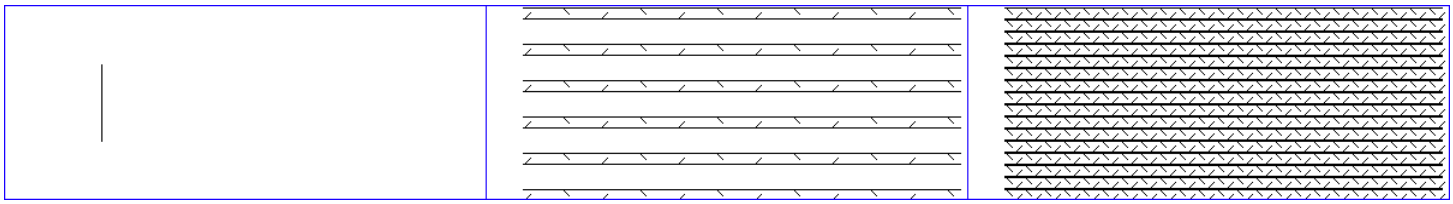


Figure 6: The three object boundaries parameters visualized. From left to right: the line object, the medium object density and the dense object density

That leaves inputs 1 and 3 as interesting inputs to use as parameters for our experimental setup.

The $H \times W$ parameter will be used in the experimental setup with the values $H \times W = \{80 \times 200, 1080 \times 1920, 3840 \times 2160\}$ as these large grid sizes resemble our goal of being able to simulate in real-time for modern high resolutions and the small grid size is chosen to analyze any special caching behavior with small grids.

The object boundaries parameter will have three different values, see Figure 6. The first is a simple vertical line with $l_H = H \cdot \frac{2}{5}$ centered at $H/2$ in the y-axis and $H/2$ in the x-axis. The second and third possible values of this parameter are similar but differ in their level of intensity. The second value is what we call "medium" object density and the third "dense" object density. For each of these densities the ratio of barriers with respect to lattice cells is dependent on the grid dimensions and is given by Equations 20, 21 and 22 which are included in the appendix. The values of these ratio's for the grid sizes in our input space are shown in Table 1.

HxW $\rightarrow$	**80x200**	**1080x1920**	**3840x2160**
$\mathbf{rat_{line}}$	$2 \cdot 10^{-3}$	$2.083 \cdot 10^{-4}$	$1.852 \cdot 10^{-4}$
$\mathbf{rat_{med}}$	$6.75 \cdot 10^{-2}$	$7.057 \cdot 10^{-2}$	$7.076 \cdot 10^{-2}$
$\mathbf{rat_{dense}}$	$2.175 \cdot 10^{-1}$	$2.578 \cdot 10^{-1}$	$2.583 \cdot 10^{-1}$

Table 1: Displays the value of Equation 20, 21 and 22 rounded to three significant digits for each of the grid dimensions in the chosen input space.

Lastly we will run all benchmarks and analyses on a single DAS-5 node that has two Intel Xeon E5-2630 v3 CPU's, with combined 16 cores, 32 hardware threads, 32KB L1 cache per core, 256KB L2 cache per core and 20MB L3 cache per CPU.

4.2 Performance metrics

Looking at the chosen research question, it includes a performance metric itself, namely the time it takes to compute a single time-step (i.e. frame) T_{frame}.

Even though this single metric would suffice to answer the research question,

several other metrics will be monitored during the performance engineering. In section 2 and 3 it is laid out that the simulation consists of four *kernels* that are executed each iteration and together produce a single next frame. Each of these kernels can be measured in terms of their latency:

- the streaming of densities - T_{stream}
- bouncing back densities on object boundaries - $T_{\text{bounceback}}$
- letting densities collide - T_{collide}
- the spawning of steady flow - $T_{\text{steadyflow}}$

With these four kernel latencies T_{frame} can be decomposed as follows:

$$T_{\text{frame}} = T_{\text{stream}} + T_{\text{bounceback}} + T_{\text{collide}} + T_{\text{steadyflow}}$$

The last performance metric to be measured is GFLOP$/s$, which is computed by dividing the total amount of floating point operations performed by the total amount of time passed thus far.

These six metrics will be used to evaluate and analyze the various implementations with the parameters defined in section 4.1.

5 Reference performance analysis

5.1 Reference implementation Analytical model

In order to gain insight in the bottlenecks of the reference implementation for the previously defined input space, which is to be used in the first performance improvement iteration, an analytical model will be created which predicts T_{frame}.

As has been explained before, a single state update can be split in a stream, bounceback, collide and a steady flow step. A full model will be formulated for each kernel. Each of those models will finally be substituted in Equation 6 to yield a model for a state update.

$$\begin{aligned} T_{\text{frame}}(W, H, F_{\text{branch}}) =& T_{\text{stream}}(W, H) + T_{\text{bounceback}}(H, W, F_{\text{branch}}) + \\ & T_{\text{collide}}(W, H) + T_{\text{steadyflow}}(H) \end{aligned} \tag{6}$$

5.1.1 Stream kernel

In the stream kernel, number density matrices are shifted row-wise or column-wise using the *roll* function. This function can be invoked to shift a matrix one row down, one row up, one column to the left or one column to the right. Each of the 8 directions are rolled into their corresponding direction (with north-east, north-west, south-west and south-east being rolled once for each component). This leads to each roll direction case to be executed three times per invocation of the stream function. To determine a model for the stream kernel a model will therefore be formulated for each case of the roll function.

```
memcpy(&buf, &(*A)[H-1][0], W * sizeof(float));
for (int y = H - 2; y >= 0; y--)
    memcpy(&(*A)[y+1][0], &(*A)[y][0], W * sizeof(float));
memcpy(&(*A)[0][0], &buf, W * sizeof(float));
```

Listing 1: The code of the roll function which shifts a matrix one row down with a wrap around.

Listing 1 is shown for the first case of the roll function. As can be seen, there are 2 memcopies of W floats outside the loop and $H - 1$ of such memcopies inside the loop. The time for this case thus equals $(H + 2) \cdot W \cdot \text{sizeof(float)} \cdot T_{\text{rollrow,bytecpy}}$. The latency of the second case of the roll function, which shifts a matrix one row upwards, can be described by same expression.

```
 for (size_t y = 0; y < H; y++) {
    float first = (*A)[y][0];
    for (size_t x = 0; x < W - 1; x++)
        (*A)[y][x] = (*A)[y][x+1];
    (*A)[y][W-1] = first;
}
```

Listing 2: The code of the roll function which shifts a matrix one column to the left with a wrap around.

Listing 2 displays the roll function for the third case. When neglecting the two statements around the inner for-loop, the latency of this case can be described as $(H - 1) \cdot (W - 2) \cdot T_{\text{rollcol,loopit}}$ where $T_{\text{rollcol,loopit}}$ is defined as the time for an inner for-loop iteration. This omission can be justified by observing that the ratio of the amount of these statements with respect to the amount of inner loop iterations $\frac{(H-1)\cdot 2}{(H-1)(W-2)} = \frac{2}{W-2}$ tends to zero quickly when the grid width increases. The latency of the final case of the roll function, which shifts a matrix one column to the right, can also be described using this expression.

The stream function as a whole can thus be modeled with equation 7 with the grid width (W) and height (H) as parameter.

$$\begin{aligned} T_{\text{stream}}(W, H) =& 6 \cdot (H + 2) \cdot W \cdot \text{sizeof(float)} \cdot T_{\text{rollrow,bytecpy}} + \\ & 6 \cdot (H - 1) \cdot (W - 2) \cdot T_{\text{rollcol,loopit}} \end{aligned} \tag{7}$$

5.1.2 Bounce back kernel

The bounceback kernel bounces any molecules that were "rolled" into an object back into the opposite direction. For each kernel invocation every grid point (y, x) in `object_grid` is checked for whether there is an object (y, x). If this is the case all densities in a certain direction are set into the density of the next

cell in the corresponding opposite direction. In Listing 3 a section of this code is shown.

```
for (unsigned y = 0; y < H; y++) {
    for (unsigned x = 0; x < W; x++) {
        if ((*object_grid)[y][x]) {
            (*grid)[EAST][y][x+1] = (*grid)[WEST][y][x];
            (*grid)[WEST][y][x-1] = (*grid)[EAST][y][x];
            // 6 similar statements...
        }
    }
}
```

Listing 3: The code of the bounceback function.

As can be seen in listing 3, there are $H \cdot W$ inner loop iterations. Each inner loop iteration contains a condition that is true when the corresponding cell is a barrier. If this is the case 8 memory statements are performed. The model of this function characterizes an inner loop iteration with it's average latency. This is an average of the time required to execute a inner loop iteration if the branch is true $T_{\text{bounce,mem}}$ and the time if this is not the case $T_{\text{bounce,cond}}$. The average is weighted with the branch condition $F_{\text{bounce,branch}}$ which is equal to the ratio of barrier to lattice cells. The total latency of the bounceback function can be modelled with Equation 8 with parameters W, H, $F_{\text{bounce,branch}}$ and to be calibrated values of $T_{\text{bounce,mem}}$ and $T_{\text{bounce,cond}}$.

$$T_{\text{bounceback}}(H, W, F_{\text{bounce,branch}}) = H \cdot W \cdot (F_{\text{bounce,branch}} \cdot T_{\text{bounce,mem}} + (1 - F_{\text{bounce,branch}}) \cdot T_{\text{bounce,cond}}) \tag{8}$$

5.1.3 Collide kernel

The collide kernel executes the computations from 2.3.3 for each grid point (y, x) and n_i. As many values from these computations can be reused within the same computations of all directions in a single grid point, the kernel loops over the grid points and then computes all the new densities for each direction. The reference implementation does not go any further in optimizing these computations than this simple reusal within the directions of a grid point. It performs 126 FLOPs per grid point, loads from 86 variables and stores into 20 variables. These numbers are calculated from the code, thus are logical FLOPs, loads and stores, not actual numbers taken from the assembly (as the compiler might have optimized away some).

The model for this kernel is formulated by Equation 9, where $T_{\text{collide},(x,y)}$ is the latency of all the operations needed to update a single grid point.

$$T_{\text{collide}}(H, W) = H \cdot W \cdot T_{\text{collide},(y,x)} \tag{9}$$

5.1.4 Steady flow

The fourth and last kernel of the simulation is the steady flow kernel. This kernel makes sure that there always exists a steady flow of fluid in the simulation. At the left most column high amounts of fluids are introduced in the east directions and low amounts in the west directions, thus creating a flow from the left to the right (as was chosen in Section 4.1).

```
// force_flows contains high amounts of molecules on
// the eastern directions and low amounts on the western directions
for (y = 0; y < H; y++) {
   for (direction in (EAST, WEST, NORTH_EAST, SOUTH_EAST,
                      NORTH_WEST, SOUTH_WEST)) {
       (*grid)[direction][y][0] = force_flows[direction];
   }
}
```

Listing 4: Pseudo-code of the steady flow kernel

In Listing 4 pseudocode is shown of the kernel. As one can see the complexity of this kernel scales with the H term, thus we formulate this kernel in our analytical model as is stated in Equation 10.

$$T_{\text{steadyflow}}(H) = H \cdot T_{\text{steadyflow},y} \tag{10}$$

5.1.5 Complete model

The complete analytical model of our simulation combines all the terms from our four kernels to find the total latency to compute a single frame, shown in Equation 11.

$$\begin{aligned} T_{\text{frame}}(W, H, F_{\text{bounce,branch}}) = {} & (6 \cdot (H+2) \cdot W \cdot \text{sizeof(float)} \cdot T_{\text{rollrow,bytecpy}} + 6 \cdot (H-1) \cdot (W-2) \cdot T_{\text{rollcol,loopit}}) + \\ & (H \cdot W \cdot (F_{\text{bounce,branch}} \cdot T_{\text{bounce,mem}} + (1 - F_{\text{bounce,branch}}) \cdot T_{\text{bounce,cond}})) + \\ & (H \cdot W \cdot T_{\text{collide},(y,x)}) + \\ & H \cdot T_{\text{steadyflow},y} \end{aligned} \tag{11}$$

5.1.6 Calibration

The to-be-calibrated parameters of the stream, bounce and collide kernels will be determined by running the reference implementation for the 1080×1920 lattice with a medium amount of barriers and 13000 iterations. More specifically, to calibrate the stream kernel parameters $T_{\text{rollcol,loopit}}$ and $T_{\text{rollrow,bytecpy}}$, the average time of these two groups of invocations will be determined. Division by the corresponding prefixes in Equation 7 will then yield the values of these parameters.

A similar process is used to determine the parameters of the collide and the steady flow kernel. First of all the average latency of each kernel is determined. Subsequently each average is divided by the corresponding prefix in either Equation 9 or Equation 10.

To determine the parameters of the bounceback kernel, timers could be placed within the loop in Listing 3. This is not desirable because this would interfere with the ILP within the loop. Another method is therefore pursued which first measures the average latency of this kernel with the same inputs as before. Subsequently the average latency is measured for the same parameters but with a high amount of barriers. If these latencies are denoted with $T_{\text{avg,medium}}$ and $T_{\text{avg,high}}$, these times are related to the to-be-determined parameters by Equations 12 and 13.

$$T_{\text{avg, medium}} = T_{\text{bounceback}}(H, W, F_{\text{bounce,branch}}) \tag{12}$$

$$T_{\text{avg, dense}} = T_{\text{bounceback}}(H, W, F_{\text{bounce,branch}}) \tag{13}$$

Setting $H = 1080$, $W = 1920$ in both equations, $F_{\text{bounce,branch}} = 7.057 \cdot 10^{-2}$ in Equation 12 and $F_{\text{bounce,branch}} = 0.26$ in Equation 13 (from Table 1) will then yield Equations 14 and 15 after simplification. As this is a system of two equations and two unknowns the desired $T_{\text{bounce,mem}}$ and $T_{\text{bounce,cond}}$ can be determined.

$$T_{\text{avg,medium}} = 2073600(0.07057 \cdot T_{\text{bounce,mem}} + 0.92943 \cdot T_{\text{bounce,cond}}) \tag{14}$$

$$T_{\text{avg,dens}} = 2073600(0.2578 \cdot T_{\text{bounce,mem}} + 0.7422 \cdot T_{\text{bounce,cond}}) \tag{15}$$

5.1.7 Verification

The previously described calibration resulted in values that can be found in Table 15 in the Appendix. To determine the accuracy of the model predictions Table 2 was generated which displays the ratio between the predicted T_{frame} and an measured average T_{frame}. As can be seen in this table the model almost yields a perfect prediction for the grid size with which calibration took place. For the smallest grid there is an overestimation of T_{frame} by $\approx 50\%$ while there is an underestimation of $\approx 10\%$ for the biggest grid. It was hypothesized that this is caused by a changing memory access time due to better or worse caching behavior.

	Line	**Medium**	**Dense**
80x200	1.46	1.45	1.52
1080x1920	1	0.99	0.98
3840x2160	0.91	0.91	0.92

Table 2: The ratio [Predicted Tframe latency] / [Measured Tframe latency] for the different gride dimensions and barrier densities that are in the chosen input space. For the 80x200, 1080x1920 and 3840x2160 dimensions [Measured Tframe latency] is an average over 10^5, 500 and 150 iterations.

To test this hypothesis `perf` was used to gather `L1-dcache-loads`, `L1-dcache-load-misses`, `LLC-loads` and `LLC-load-misses` for each input in the input space. The amount of L2 requests and hits could not be gathered due to unreliable HW counters for the E5-2630V3 CPU (as can be read in the perf manual). By assuming that $\text{Requests}_{L2} = \text{Misses}_{L1}$ and $\text{Misses}_{L2} = \text{Requests}_{L3}$, the *average access time* (AAT) was determined for each input in the input space. The AAT can be calculated according to equations 16, 17 and 18. Every hitrate was determined from the gathered hit and miss rates while each memory load latency was obtained by running lmbench. The specific values and their origin are shown in Figure 12 which is included in the appendix.

$$\text{AAT} = \text{Hitrate}_{\text{L1}} \cdot \text{Lat}_{\text{hit,L1}} + (1 - \text{Hitrate}_{\text{L1}})\text{Lat}_{\text{miss,L1}} \tag{16}$$

$$\text{Lat}_{\text{miss,L1}} = \text{Hitrate}_{\text{L2}} \cdot \text{Lat}_{\text{hit,L2}} + (1 - \text{Hitrate}_{\text{L2}})\text{Lat}_{\text{miss,L2}} \tag{17}$$

$$\text{Lat}_{\text{miss,L2}} = \text{Hitrate}_{\text{L3}} \cdot \text{Lat}_{\text{hit,L3}} + (1 - \text{Hitrate}_{\text{L2}})\text{Lat}_{\text{miss,L3}} \tag{18}$$

The application of this procedure resulted in Table 3. As can be seen the AATs of the 80×200 grid are approximately 55% higher than that of the 1080×1920 grid. Furthermore the AATs of the 3840×2160 grid are approximately 8% lower than that of the 1080×1920 grid. As these differences coincide with the deviations that have been found this makes differing cache behavior a very plausible reason for the model inaccuracies.

	Line	**Medium**	**Dense**
80x200	1.45	1.45	1.46
1080x1920	2.23	2.26	2.33
3840x2160	2.46	2.49	2.49

Table 3: The AAT in ns for each (grid size, barrier density) combination in the chosen input space.

5.2 Kernel profiling

The next step that was taken to analyze the performance of the reference implementation was a profiling of the four kernels, that is measuring the time each kernel takes relative to T_{frame}. In Figure 7 the benchmark results for this analysis are shown.

A few patterns appear in these results. Firstly these results show that the steady flow kernel is almost negligible, especially for the larger resolutions. This can be easily attributed to the fact that its complexity only scales with H, as can be seen in the analytical model in Equation 11, thus disappearing from the big picture when the resolution increases. Secondly it is shown that the collide kernel takes in all configurations the most time of all kernels, by a big margin. Thirdly the results show that the bounceback kernel increases in its relative latency as the amount of grid points that contain an object is increased. This is logical when looking at the model because $F_{\text{bounce,branch}}$ increases which

causes the average inner loop latency to increase due to the fact that processing a barrier is more time-intensive than not doing so.

Lastly the stream kernel's relative latency appears to increase with the increase in resolution, however the difference between the 1080×1920 and 2160×3840 resolution is quite small. This seems to indicate that at small resolutions, this solely memory heavy kernel, seems to be relatively fast. A possible cause of this is that small grid sizes fit better inside the cache. More specifically, the 80×200 takes up 704 kb [1] and thus fits in the L3 cache while the other grid sizes do not. The previous analysis using the AATs, which are shown in Table 3, also strengthen this suspicion.

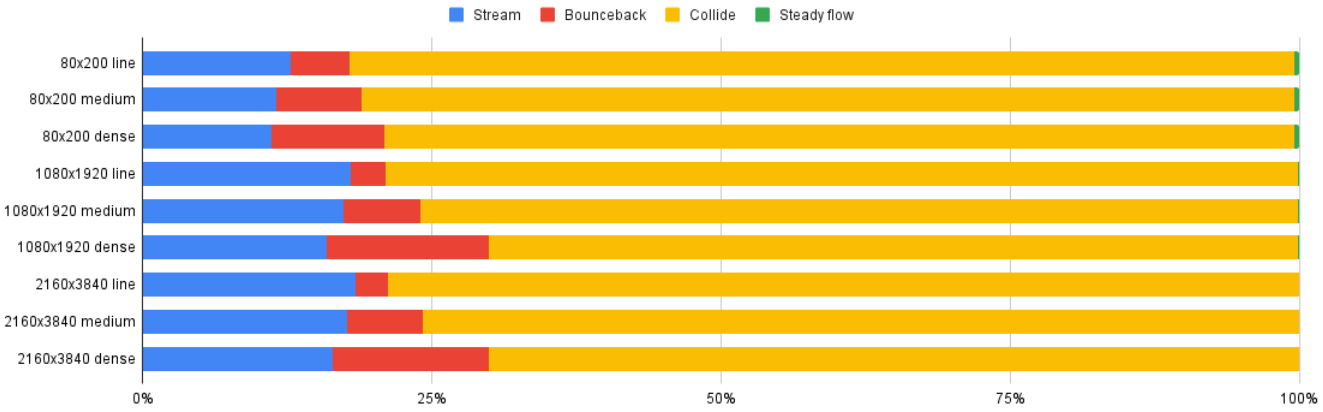


Figure 7: Kernel profiling of the reference implementation, showing the percentage of T_{frame} that each kernel takes up for each configuration.

6 Performance improvement iterations

6.1 Vectorization of the collide kernel

It has been noted previously that the collide kernel is dominant in T_{frame}. The first performance iteration will focus on this kernel because the achievable speedup of T_{frame} will then be maximized. Recall that the collide kernel collides molecules after they have been moved to new positions.

From a computational perspective, the first C implementation of this kernel can be described with the pseudocode shown in Listing 5. An inner x loop iteration is responsible for determining the 9 different number densities per grid point. As can be seen here, a new number density of a grid point, is only dependent on it's previous value and some constants.

[1] The 9 number density arrays as well as the rho, ux and uy array take up $11 \cdot 80 \cdot 200 \cdot \text{sizeof(float)} \cdot 10^{-6} = 0.704$ MB.

```
for y in 0 .. H -1:
   for x in 0 .. W - 1:
        rho, ux, uy = 0
        for dir in directions:
            rho += ndens[dir][y][x]

        for dirx, diry in [(EAST, NORTH,), (NORTH_EAST, NORTH_EAST), (SOUTH_EAST,NORTH_WEST)]:
            ux += ndens[dirx][y][x] / rho
            uy += ndens[diry][y][x] / rho
        for dirx, diry in [(WEST, SOUTH), (NORTH_WEST, SOUTH_EAST), (SOUTH_WEST,SOUTH_WEST)]:
            ux -= ndens[dirx][y][x] / rho
            uy -= ndens[diry][y][x] / rho

        for dir in directions:
            temp = 4.5 * (ex[dir] * ux + ey[dir] * uy)^2 + 3*(ex[dir] * ux + ey[dir] * uy)
                    - 1.5 * (ux^2 + uy^2) + 1
            ndens[dir][y][x] = (1 - omega) * ndens[dir][y][x] + rho * omega * w[dir] * temp
```

Listing 5: Highly simplified pseudocode of the collide kernel. In this pseudocode ex and ey are defined as the x and y components of the direction vectors of the DSQ9 grid (see Figure 1). The array w consists of constants which can be found in Equation 2.

As a 256-bits SIMD register (maximum supported on the DAS-5) allows 8 single precision floats, it is possible to determine the number densities of 8 different grid positions simultaneously using vectorization. Even though it could be possible that the collide kernel is memory bound, there is no clear path towards a memory optimization because of the constraints of the Lattice Boltzmann method. For example techniques such as cache blocking will not yield any benefit due to the fact that there is no data reusal of number density elements within a call of the collide kernel. Vectorization will be therefore be the first attempted optimization regardless whether the kernel is memory bound or compute bound.

6.1.1 Speedup prediction with vectorized collide kernel

In order to predict a performance increase for the three considered grid dimensions when vectorization were to-be-performed, *integrated roofline models* (IRMs) were generated for the collide kernel using Intel Advisor. An IRM consists of a L1, L2, L3 and DRAM AI whereby each AI is the ratio #FLOP / bytes. [2]Furthermore bytes refers to the traffic between that memory and memory "higher" in the hierarchy or the core.

As the number of operations in the collide kernel is independent of the amount of boundaries, it is likely that an IRM will be independent of its value. There might be an interference due to caching behavior in the stream and bounceback kernels, but it is assumed this effect is negligible. This parameter was therefore arbitrarily set to dense in the to-be-generated IRMs.

Some corrections were made to the generated IRMs using Intel Advisor. First of all variation of peak compute and peak memory bandwidth bounds were

observed across various runs. This variation is to be expected because benchmarks are used internally to determine these values. Therefore corresponding maximum values are used (n=7).

It was also observed that for the different grid sizes, the absolute performance of the collide kernel, reported by Intel Advisor, differ from manually obtained values. These values were manually obtained with $(91 \cdot H \cdot W)/T(H, W)$ where $T(H, W)$ is the average latency of the kernel for a certain grid size, and 91 FLOPS occur per grid point. The latter value was found by evaluating the generated assembly code. While this initially seemed to stem from a different $T(H, W)$ due to overhead, a further investigation resulted in observing that Intel Advisor also used a different FLOP count. Therefore the manual obtained values are substituted in each IRM. As each FLOP count is also used in determining the AIs (#FLOP / bytes), these values were corrected as well.
Figure 8 show the generated IRMs for each grid size. It can be seen that the reference implementation's absolute performance is consistently below the scalar add peak. The highest roof that is reachable when vectorization were to be performed, is indicated with a dotted line for each IRM. It can therefore be seen that the collide kernel is eventually memory bound. More specifically, the kernel is bound by L3 bandwidth for the 80x200 grid while it is bound by DRAM bandwidth for the other grid sizes. This is logical because the 80x200 working set of nine different number density arrays takes up $80 \cdot 200 \cdot 9 \cdot \text{sizeof(float)} \cdot 10^{-6} =$ 0.576MB, fitting fully in the 20MB L3 cache, while the bigger working sets take up 74MB and 298MB.

It can furthermore be seen that the absolute performance of the collide kernel decreases for bigger grid sizes. The absolute performance is 2.84GFLOP/s (80x200), 1.91GFLOPS (1080x1920) and 1.69 GFLOPS (3840x2160). The decrease between the 80x200 and 1080x1920 grid is to be expected because of the introduction of DRAM requests which cause slower memory access times. Even though Table 4 shows that the DRAM AI become slightly lower between the 1080x1920 and 3840x2160 grid the latter difference does not seem to be fully warranted. When running `likwid-perfctr` on both the 1080x1920 and 3840x2160 grid size with dense boundaries and the group `CYCLE_STALLS`, the execution stall rate of the collide kernel increases from 24% to 33%. In both cases, 98% of these stalls were caused by memory loads.[2]It therefore seems plausible that the performance difference is caused by a lower DRAM AI.

AI→	**L1 AI**	**L2 AI**	**L3 AI**	**DRAM AI**
80x200, dense, 7.5E4 iterations	0.555	0.946	0.951	-
1080x1920, dense, 500 iterations	0.555	0.952	0.949	0.952
3840x2160, dense, 100 iterations	0.555	0.953	0.950	0.950

Table 4: The AI values for every generated IRM displayed in figure 8.

[2]The collide kernel was isolated by using the Likwid Marker API.

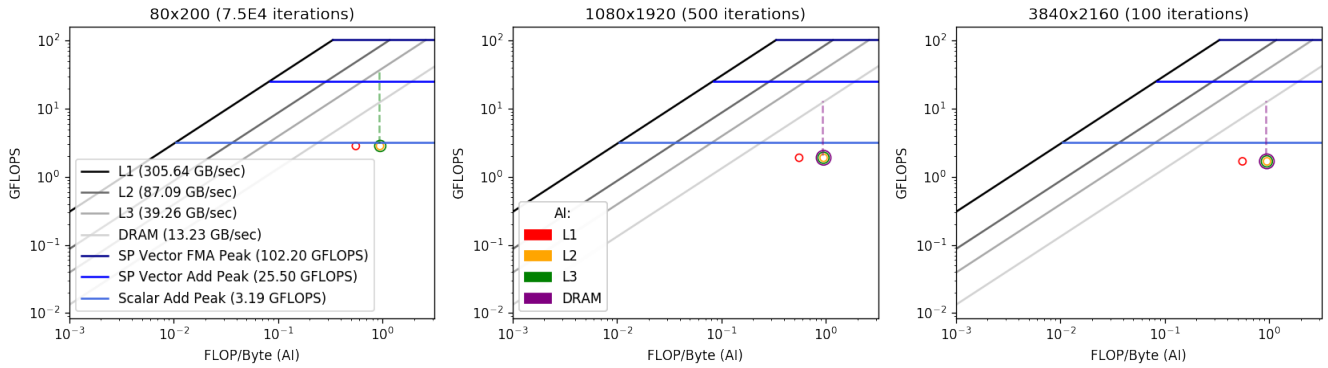


Figure 8: (a),(b),(c) from left to right. Each figure is a IRM of the collide kernel in the reference C implementation generated by Intel Advisor where the program was ran for the denoted grid size and amount of iterations with dense boundaries. Absolute performances of the collide kernel were manually determined and each AI was corrected with the correct FLOP count. Additionally, memory bandwidths and peak compute bounds are maximum values (n=7).

Given these current absolute performances and the maximum achievable performances according to the IRMs maximum speedups were calculated for the collide kernel which are shown in table 5.

	80x200	**1080x1920**	**3840x2160**
Current abs performance	2.84 GFLOP/s	1.91 GFLOPS/s	1.69 GFLOPS/s
Maximum abs performance	[25.5, 37.33] GFLOPS/s	12.59 GFLOPS/s	12.57 GFLOPS/s
Maximum speedup	[8.98, 13.14] ×	6.61 ×	7.44×

Table 5: Current and predicted maximum performance and maximum speedup. Note that the upper bound is a range for the 80x200 grid dimensions whereby the low end corresponds to the peak vector add performance while the high end corresponds to L3 bounded performance when FMA units are utilized.

To determine a overall maximum speedup for T_{frame}, average kernel times were obtained for all grid dimensions and barrier density combinations in the input space with 10^5, 500 and 150 iterations for the 80x200, 1080x1920 and 3840x2160 grid. Given the values for a certain input parameter combination, the maximum speedup can be calculated with $(T_{\text{other}} + T_{\text{collide}})/(T_{\text{other}} + T_{\text{collide}}/S)$ where T_{other} denotes the sum of all average kernel times other than that of the collide kernel and S refers to the corresponding collide kernel speedup shown in Table 5. The resulting values are shown in Table 6.

	line	medium	dense
80x200	5.40	5.12	4.82
1080x1920	4.20	3.97	3.55
3840x2160	4.42	4.19	3.75

Table 6: The predicted overall maximum T_{frame} speedup of the vectorized implementation with respect to the reference implementation.

6.1.2 Implementation of vectorized collide kernel

The vectorization was implemented using Intel intrinsics provided by *immintrin.h*. For vectorizing the collide kernel two options were considered: (1) vectorize over multiple grid points and compute 8 grid points in the inner loop, or (2) vectorize over the many FLOPs that are needed for a single grid point and compute multiple directions for a single grid point in the inner loop. Although the second option sounds attractive, even with the many FLOPs needed for a single iteration, there are not enough to keep the FP execution units optimally occupied.

For example the u_x and u_y terms from Equation 3 and 4 are needed for every direction i in n_i^{new}. But there is no way to fill and fully utilize the 8 FP32 wide execution units, as with this computation the operator precedence prevents the formulas of being split up into more than 2 FLOPs that can be executed in parallel, thus creating a bottleneck. While with option 1, this problem does not arise, as for 8 grid points the 8 u_x's and 8 u_y's are both computed in parallel. The second option also has the advantage of being significantly less complex (albeit still quite complex) as the formulas don't have to be dissected into parallelizable FLOPs. Therefore it was decided to vectorize over 8 grid points in a single inner loop.

6.1.3 Benchmarking vectorized collide kernel

To evaluate the performance of the SIMD implementation, the average latency of the collide kernel was measured for all parameters in the input space with $7.5E4$ (80x200), 500 (1080x1920) and 100 (3840x2160) iterations. These values combined with the measurements for the reference implementation result in the speedups which are displayed in Table 7. For the 80x200 grid dimensions, the speedup prediction is sound, as the percent error between the predicted and realized speedup is always below 10%. For the other two grid sizes the realized speedups exceed the predictions.

Although the maximum DRAM bandwidth was taken over various Intel Advisor runs, large variance was observed in the values. It could therefore be that the DRAM bandwidth, that was used in the speedup predictions, was underestimated. If the theoretical maximum DRAM bandwidth of a single E5-2630V3 core were used (14.928 GB/s[3]), the predicted maximum speedup is 7.44[4]

[3] $1.866 \cdot 8 = 14.928$ GB/s with 1.866 GHZ as maximum memory speed and a bus width of 8 bytes.

[4] $(0.952 \cdot 14.928)/1.91$

(1080x1920) and 8.39 [5] (3840x2160). All realized speedups are lower than these estimates, which indicates that this is a plausible reason for the prediction error. The overall speedup of T_{frame} in the vectorized implementation with respect to the reference implementation is at most 5.13 and at least 3.62. The percent error of these speedups with respect to the predicted speedups (Table 5), that were corrected for the higher DRAM bandwidths, are shown in Table 9. These errors are small and reside between -2.68% and -5.00%.

	line	**medium**	**dense**	**Predicted**
80x200	8.33	8.13	8.37	$\leq \times[8.98, 13.14]$
1080x1920	6.78	6.71	6.78	$\leq \times 6.61$
3840x2160	7.57	7.54	7.57	$\leq \times 7.44$

Table 7: Displays the speedup of the average T_{collide} of the vectorized implementation with respect to the reference C implementation together with the original predictions.

	line	**medium**	**dense**
80x200	5.13	4.76	4.57
1080x1920	4.30	4.02	3.62
3840x2160	4.54	4.26	3.84

Table 8: The overall speedup of the average T_{frame} of the vectorized implementation with respect to the reference C implementation.

	line	**medium**	**dense**
80x200	-5.00%	-6.99%	-5.20%
1080x1920	-4.27%	-4.83%	-3.10%
3840x2160	-3.63%	-3.86%	-2.68%

Table 9: Displays the percent error between the predicted and realised overall vectorized T_{frame} speedup with respect to the reference implementation for each element in the input space. Note that the predicted speedup was corrected with the theoretical DRAM bandwidth.

[5] $(0.950 \cdot 14.928)/1.69$

6.1.4 Performance analysis vectorized collide kernel

To analyze the slight discrepancy between the predicted speedup and the achieved speedup *llvm-mca* was used to investigate potential bottlenecks. *llvm-mca* uses a theoretical model of a given microarchitecture (Intel Haswell in our case) to simulate the CPU with its pipeline and execution units during execution of a certain section of assembly (the inner loop of the collide kernel in our case).

Looking at the assembly itself it can be seen that some registers are spilled back onto the stack. This is something that the compiler inserts into the assembly when there are not enough registers for the data, preventing the need to compute the same information twice.

The bottleneck analysis simulation showed that the resources (i.e. execution units) are in $\sim 56\%$ of the cycles during an inner loop "pressured". Meaning that atleast some slowdown occurs because operations are delaying each other as there are not enough resources to achieve optimal floating-point execution unit occupancy. In $\sim 70\%$ of the cycles, data dependencies are preventing the executions to be optimally occupied. For example the u_x and u_y values are needed for all subsequent computations, but inside the u_x and u_y computations data dependencies prevent reaching optimal GFLOP/s.

6.2 Improvement upon the vectorized collide kernel

The previous performance iteration showed speedups that slightly underperformed from the perspective of the predictions. The subsequent analysis provided a path toward improving bottlenecks in the vectorization. Therefore this (small) performance iteration will focus on improving these bottlenecks.

As improving resource pressure is very difficult in this kernel with solely floating-point operations, it was chosen to pursue improving data dependencies, especially as this is a bigger factor than resource pressure. Iterating once more upon vectorization is not expected to give speedups that fully align with the predictions made with the first vectorized implementation, as some data dependencies are inherent in the algorithm.

Firstly one of the two divisions by ρ was removed by doing one division $\frac{1}{\rho}$ and then doing two multiplications, arriving at the same result with less latency. By carefully examining the code and dependencies between computations, a few more computations were able to be switched around which should result in less data dependencies.

6.2.1 Benchmarking of the improved vectorized collide kernel

Now performing the benchmarking again we find the following results shown in Table 10 and Table 11. These numbers show that the speedups only occur with the small resolution with this improved version. Looking at the speedup of a complete frame with the two larger resolutions, there is even a very slight slowdown.

	line	medium	dense
80x200	1.15	1.89	1.56
1080x1920	1.00	1.01	1.01
3840x2160	1.00	1.01	1.01

Table 10: The speedup of T_{collide} of the improved vectorization implementation with respect to the original vectorization implementation.

	line	medium	dense
80x200	1.08	1.89	1.56
1080x1920	0.99	0.99	1.00
3840x2160	0.98	0.99	0.99

Table 11: The speedup of T_{frame} of the improved vectorization implementation with respect to the original vectorization implementation.

6.2.2 Performance analysis improved vectorization

To investigate how and why the improved vectorization does not benefit the configurations with the two larger resolutions, performance counters and *llvm-mca* were used to analyze the runtime characteristics and generated assembly.

When comparing the *llvm-mca* bottleneck analysis of the improved vectorization to that of the original vectorization, the resource pressure drops from $\sim 56\%$ to $\sim 38\%$. The percentage of cycles with data dependencies increases from $\sim 70\%$ to $\sim 75\%$. Especially the slight increase in the data dependencies is interesting, as that is the area the improvement targeted. As these results are very hard to dissect (requiring deep assembly inspection), it was chosen to analyze through other approaches instead of dividing further into these results.

Inspecting the assembly of the original vectorization and the improved vectorization, it is found that the total amount of assembly instructions in the inner loop decreases from 112 to 105 with the optimization, the instructions that contain a memory load or store μop decreases significantly from 53 to 45. Indicating that there should atleast be some kind of performance difference measurable concerning memory behavior. This inspection of the assembly also resulted in the discovery that the two divisions in the original vectorization were already optimized away to a single *RCPPS* (computes the reciprocal) and two multiplies.

Many performance counters were measured using *likwid* while executing the collide kernel for the 80×200 and 1080×1920 resolutions (recall Table 11). Only the group of resulting metrics that showed any novel results was caching, these metrics are shown in Table 12. Two metrics stand out with a significant difference between the two implementations. Firstly the L3 miss ratio increases slightly with the improved vectorization for both resolutions, while the stall rate (the % of cycles that are wasted because of waiting for loads) decreases with 6.8% for the small resolution but increases by 2.3% for the larger resolution. Indicating that the changes that were observed in the assembly seem to only

	80×200 original	80×200 improved	1080×1920 original	1080×1920 improved
L2 rate	0.234	0.238	0.338	0.34
L2 miss ratio	0.214	0.212	0.199	0.198
L3 miss ratio	0.015	0.042	0.129	0.16
Stall rate (caused by loads)	28.2%	21.4%	48.5%	50.8%

Table 12: Measured metrics (calculated from measured performance counters) for the 80×200 and 1080×1920 resolutions with the original vectorization and the improved vectorization.

positively affect the small grid sizes. Unfortunately the performance counters did not give any insight to why this only occurs with the small grid.

It is hypothesized that when the grid does not fit into L3, the new ordering of computations and the long latencies on the fewer memory loads, lead to executions where more data dependencies occur than with the original vectorization. Thus still leading to worse performance with larger grid sizes.

6.3 Parallelization of the collide kernel

As the improved vectorization iteration gave a slight slowdown with the two larger resolutions, the original vectorization will be used in subsequent iterations. To see which kernel should be the focus for this performance iteration, kernel profiling is performed on this vectorized implementation. In Figure 9 it is clearly shown that although the dominance of the collide kernel has decreased, it is by a significant margin still dominant. As there is no realistic path towards optimizing the inner-loop, the next logical step is to parallelize the loop of the collide kernel.

6.3.1 Speedup prediction with parallel collide kernel

To limit complexity the range of T will be limited to $T = \{2, 4, 8\}$ throughout the report when analyzing parallelism.

The analytical model of the collide kernel in Equation 9, computes the total latency of the kernel by calculating the amount of inner loop iterations and multiplying that by the latency of a single inner loop iteration. When parallelized with T threads, these $H \cdot W$ amount of inner loop iterations can be divided over the threads and executed in parallel, thus giving the predicted speedup of $\times T$ and the analytical model formula:

$$T_{\text{collide}}(H, W) = \frac{H \cdot W \cdot T_{\text{collide},(y,x)}}{T} \tag{19}$$

However this prediction is quite naive as this does not take into account that the vectorized version is most likely still memory-bound, as no explicit memory optimizations were made. Therefore roofline models will be generated and used to predict the parallel speedup.

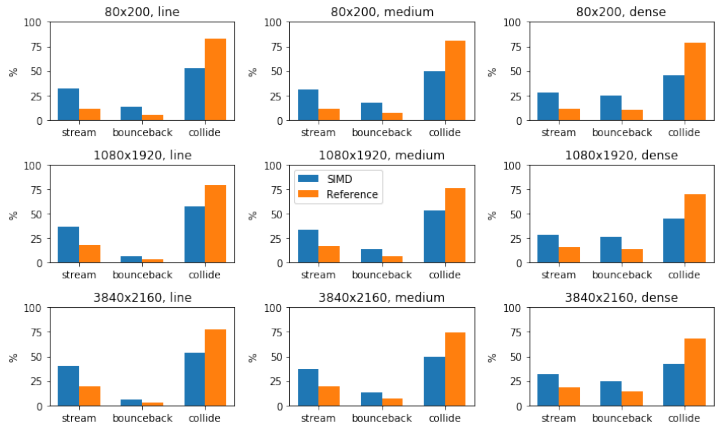


Figure 9: Kernel profiling of the vectorized collide kernel, showing the percentage of the average T_{frame} that each kernel takes up for each configuration. For the 80x200, 1080x1920 and 3840x2160 grid size 7.5E4, 500 and 100 iterations were used respectively.

To generate these roofline models without having knowledge about the code, it has to be assumed that the AI does not change when moving towards the parallel implementation (i.e. using the measured AI from the vectorized implementation in the parallel roofline models). As with this assumption the other factors in the model (i.e. maximum theoretical bandwidths and maximum theoretical GFLOP/s) can be calculated from the amount of threads. Using this assumption and the known GFLOP/s numbers for the vectorized implementation, the roofline models were generated for all threads counts T, shown under Appendix 8.3 in Figure 13. Here we see that once more that only the small resolution is bound by L3 cache (the purple DRAM dot is completely outside of the graph as the whole grid fits into L3 cache), while the two larger resolutions are bound by DRAM. Using these roofline models the predicted speedups can be calculated for the collide kernel, shown in Table 13.

threads →	**2**	**4**	**8**
80x200	1.49	2.99	5.97
1080x1920	2.53	4.51	4.84
3840x2160	2.56	4.56	4.89

Table 13: The predicted speedup of the parallelized collide kernel with respect to the vectorized collide kernel for all considered thread count grid dimension combinations. Note that each prediction is independent of the boundary density setting.

```
#pragma omp parallel for schedule(static) \
                         firstprivate(H, W, grid, rho_array, u_array)
for (size_t y = 0; y < H; y++) {
    for (size_t x = 0; x < W; x+=8) {
        // Vectorized FLOPs ...
```

Listing 6: Parallelized version of the collide kernel loop

6.3.2 Implementation of the parallel collide kernel

OpenMP was used to parallelize the collide kernel, as this kernel is a on a high level a simple loop that can be divided over threads.

To implement the parallelization of the collide kernel, OpenMP was employed with a *parallel for* pragma as can be seen in Listing 6. The two loops are not collapsed as this would pose problems for the vectorization and not give any gains with the already high granularity of a single row. The scheduling is simply *static* as the workload of each row is always the same. There is no *if*-clause on this *parallel for* as it was empirically determined that even with the smallest resolution, a speedup was still achieved going from single-threaded to parallel with 2 threads.

6.3.3 Benchmarking the parallel collide kernel

Running the benchmarking on the parallel collide kernel we find the collide speedups shown in Figure 10, the overall speedups shown in Table 14 and the kernel profiling found in Figure 11. The collide speedups are compared with those predicted in Section 6.3.1. This figure shows that almost completely across the board, the implementation does not come close to reaching the predicted speedups.

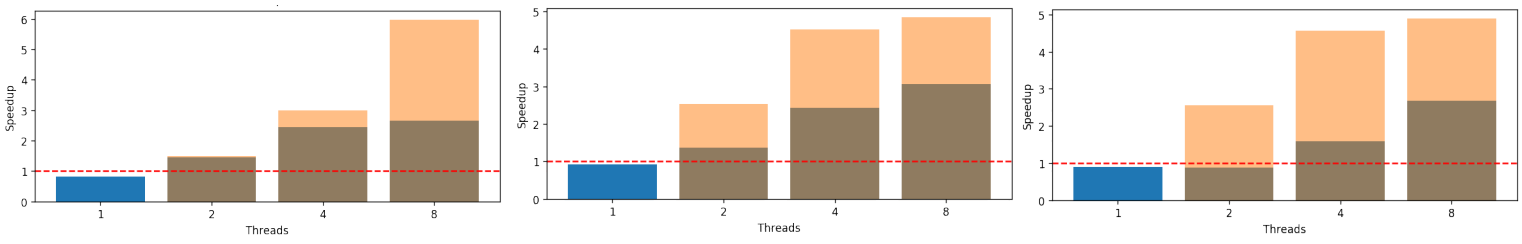


Figure 10: The speedup of the parallelized collide kernel with respect to the vectorized collide kernel compared to the prediction (all configurations used the line object density). Blue above brown means underestimation and orange above brown means overestimation.

	line	medium	dense
80x200	5.52	5.11	4.78
1080x1920	6.79	6.21	5.15
3840x2160	6.64	6.10	5.18

Table 14: The speedup of the average T_{frame} of the parallelized collide kernel with respect to the reference implementation with $T = 8$ (highest performing thread count). Note that 7.5E4, 500 and 100 iterations were used for the 80x200, 1080x1920 and 3840x2160 resolution respectively.

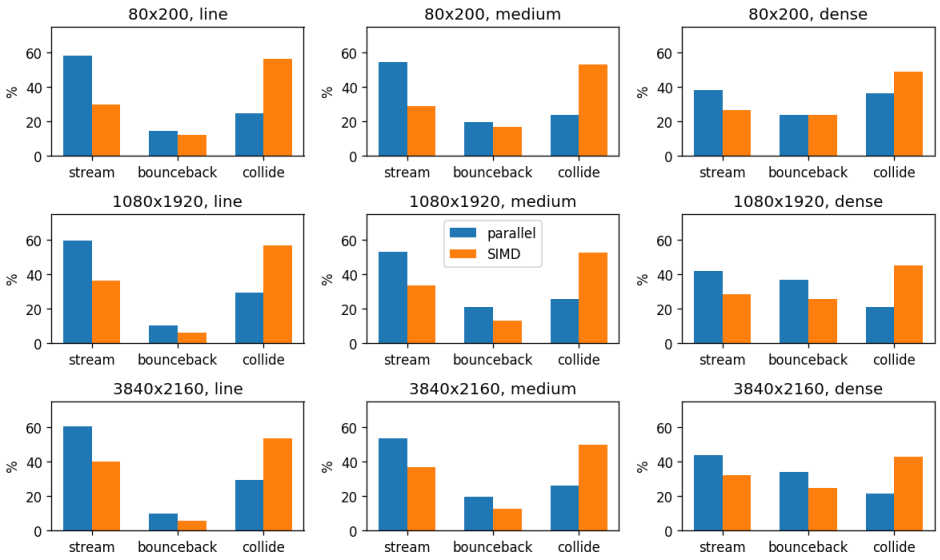


Figure 11: Kernel profiling of the parallel implementation compared to the vectorized implementation. Note that an average latency is used for each kernel across 7.5E4 (80x200), 500 (1080x1920) and 100 (3840x2160) iterations. The steadyflow kernel is omitted because it is not significant.

6.3.4 Performance analysis of the parallel collide kernel

As this performance difference between the speedup predictions from the roofline models and the realized speedup is quite large, analysis will be performed to determine the cause of this difference.

In the predictions it is assumed that the AI of the parallel implementation is the same AI across all threads as the AI in the vectorized implementation. Because of this the first avenue pursued was analyzing the memory behavior. It was hypothesized that the fact that the main thread modifies the complete grid in the stream kernel, storing these results in its local cache and invalidating local caches of the other cores, leads to the OpenMP threads having a completely empty cache when they are able to execute the collide kernel on their portion

of the grid again.

However when measuring the various memory and caching related performance counters, no significant differences between the vectorized implementation and parallel version were found. Measuring the caching behavior for the parallel implementation with Intel Advisor (with adjusted FLOP counts as Intel Advisor measures these wrong), we find a very slight decrease < 0.05 in all AI's except for the 200×80's L3 AI that increases 8-fold (only tested with 8 threads). This 8-fold increase in L3 AI for the small resolution is thought to occur because of the $8\times$ increase in total L1 and L2 cache size, leading to fewer L3 reads when the grid fits better into these smaller caches. However these findings do not get us any closer to the cause of the performance discrepancy and reject our earlier hypothesis.

When the clock performance counters were measured, the clock frequencies were found to differ vastly per core throughout the various thread counts and resolutions. It was found that core 0 (the core where the main thread with the stream and bounceback kernels execute) in all configurations runs at a higher clock frequency than the other cores. In some cases core 0 has a clock speed twice that of the other cores.

Why this clock frequency imbalance occurs is easy to trace, the OpenMP threads suspend after they are done with their collide kernel workload and their respective cores clock down to a low clock frequency (likely being the minimum clock frequency of 1.2Ghz on the DAS-5 node) while core 0 stays active working on the other kernels. As a result its clock frequency remains high (between the base frequency and maximum turbo frequency depending on the amount of cores in use). Because the collide kernel only takes up a small fraction of a second there isn't enough computation time to clock up the cores that are suspended while core 0 executes the other kernels. Manually timing how long each thread spends inside of the compute section of the collide kernel confirms this theory, as the cores with the low clock frequency take significantly longer to complete their work.

Because the roofline models used for the speedup predictions, assume that all the threads run at their maximum clock frequency (depends on the amount of cores in use), the observed prediction errors seem plausible. Unfortunately restrictions on the DAS-5 do not allow pinning the CPU clock frequencies at their maximum clock frequency to test whether this would make the predictions line up with the achieved performance.

7 Conclusion

In this report the optimization process of a 2D Lattice Boltzmann Fluid simulation has been documented. A C port of a Python implementation served as a starting point for this process. After kernel profiling it was discovered that the collide kernel was dominant in the latency of a single fluid state update (T_{frame}). Therefore the subsequent steps focused on this kernel. The first performance iteration vectorized the collide kernel with Intel Intrinsics, after which

the second iteration attempted at further improving this vectorization through analysis, but did not improve upon original vectorization. The third and final performance iteration parallelized the collide kernel using OpenMP. These steps combined yielded a speedup between 5.11 and 5.18 of the average T_{frame} with respect to the reference implementation, with the value of T_{frame} being heavily reliant on the parameter configuration used. This is our current answer to the research question:

> How fast can a single time-step in the 2D fluid dynamics simulation be performed?

Achieving a better performance is feasible, this is clearly shown by the discrepancy between the predicted and realized parallel speedups as well as other possible improvements which will be discussed in section 7.2.

7.1 Lessons learned

During this performance engineering process several lessons were learned. Firstly the results from tools, such as generated models, benchmarks results and performed assembly analysis, should not be taken be taken as complete truths at face value.

For instance we found that Intel Advisor reported inaccurate DRAM bandwidth limits and that some performance counters (especially when needing to resort to using *perf* to measure these when using the DAS-5) reported numbers that were flat out wrong when manually verified. Secondly we struggled with keeping different compiler versions and compiler flags in check.

The DAS-5 provides GCC 9.3, one student had GCC 11.1 on their machine and another student had GCC 5.4 on their machine. We found that to keep analysis (especially when using the generated assembly) consistent, comparing results between compiler versions lead to misleading results.

Changing compiler flags in a certain iteration also turned out to be dangerous as forgetting to change them back invalidated results on that iteration, as comparison with an earlier iteration would not be fair anymore.

Lastly we experienced that not pinning threads on specific cores lead to analysis and benchmarking results that are misleading. Due to the fact that our problem is very memory-bound, threads switching cores leads to performance penalties in caching that are very visible in the results. After this was determined to be the cause of some very unexpected results, all benchmarks and analyses were re-performed using pinning.

These factors combined lead to a misprediction of roughly -50% instead of roughly $[-7, -3]\%$ in Section 6.1.3 with the first vectorization implementation.

7.2 Future work

From the kernel profiling of the last performance iteration, it was concluded that the stream kernel is now the dominant factor in all configurations. For the dense boundary setting, the bounceback kernel is only a few percents less

dominant than the stream kernel. Although we did not have the opportunity to optimize these kernels due to time constraints, we did have several ideas about further performance improvements. These will be shortly described here.

The stream kernel simply moves all molecules into their respective direction, with several loops that perform *memcpy()*'s. However we think that this kernel can be practically eliminated by instead of moving the molecules inside of the grid, keeping track off an offset. Take for example the down direction. Instead of copying all the molecules from (y, x) to $(y+1, x)$, the point that is seen as the start of the grid $(y_{\text{up}}, x_{\text{up}})$ is changed to $(y_{\text{up}} + 1, x_{\text{up}})$ in each iteration. When other kernels then want the molecules in the up direction at logical index (y, x) they look in the buffer at position $(y_{\text{up}} + y, x_{\text{up}} + x)$.

The bounceback kernel could be optimized by instead of having a $H \times W$ object grid that denotes whether on a given (y, x) an object is present, a vector of (y, x) indices could be used wherein the presence of element (y, x) denotes that on this position an object grid is present.

The performance of the current collide parallel implementation could be improved by pinning the CPU clock frequency. Further improvements could be gained by parallelizing the stream kernel as well as the bounceback kernel. Parallelizing the other kernels too would prevent the down clocking of the non-thread 0 cores, caused by these cores having to suspend while waiting for thread 0 to complete the steady flow, stream and bounceback kernels. Removing the need for pinning the CPU clock frequency, while still providing the collide kernel with more performance.

8 Appendix

8.1 Barrier lattice cell ratio's for line/medium/dense

$$\text{rat}_{\text{line}}(H, W) = \frac{\lfloor 3H/10 \rfloor \lfloor 7H/10 \rfloor}{HW} \tag{20}$$

$$\text{rat}_{\text{med}}(H, W) = \frac{\left\lceil \frac{1}{30}(H-2) \right\rceil \left(2W - 70 + 5 \left\lceil \frac{1}{32}(W-35) \right\rceil\right)}{HW} \tag{21}$$

$$\text{rat}_{\text{dense}}(H, W) = \frac{\left\lceil \frac{1}{10}(H-11) \right\rceil \left(2W - 70 + 5 \left\lceil \frac{1}{8}(W-35) \right\rceil\right)}{HW} \tag{22}$$

8.2 Analytical model reference implementation

$\mathbf{T_{rollrow,bytecpy}} = 0.088720$	$\mathbf{T_{rollcoll,loopit}} = 0.407201$	$\mathbf{T_{bounce,mem}} = 12.9577$
$\mathbf{T_{bounce,cond}} = 0.899223$	$\mathbf{T_{collide,(y,x)}} = 20.028574$	$\mathbf{T_{steadyflow,y}} = 64.750737$

Table 15: The calibrated values of the reference analytical model in ns.

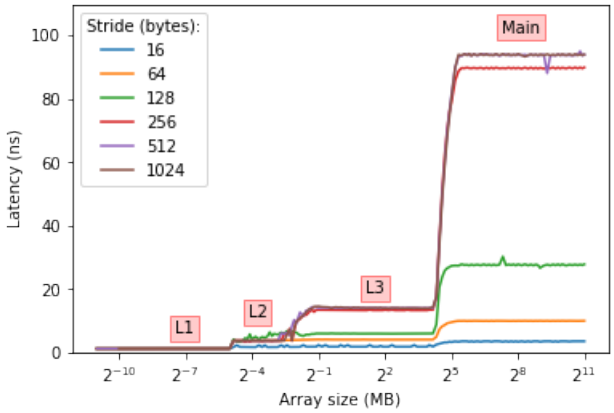


Figure 12: A figure generated by lmbench for a DAS5-node from which the memory load latency of each memory type can be derived. The specific values are as follows: $\text{Lat}_{\text{L1}} = 1.25\text{ns}$, $\text{Lat}_{\text{L2}} = 3.7\text{ns}$, $\text{Lat}_{\text{L3}} = 14\text{ns}$ and $\text{Lat}_{\text{main}} = 94\text{ns}$. They were obtained by determining the height of each of the 4 plateau's in the graph with a stride of 1024 bytes.

8.3 Parallel roofline prediction models

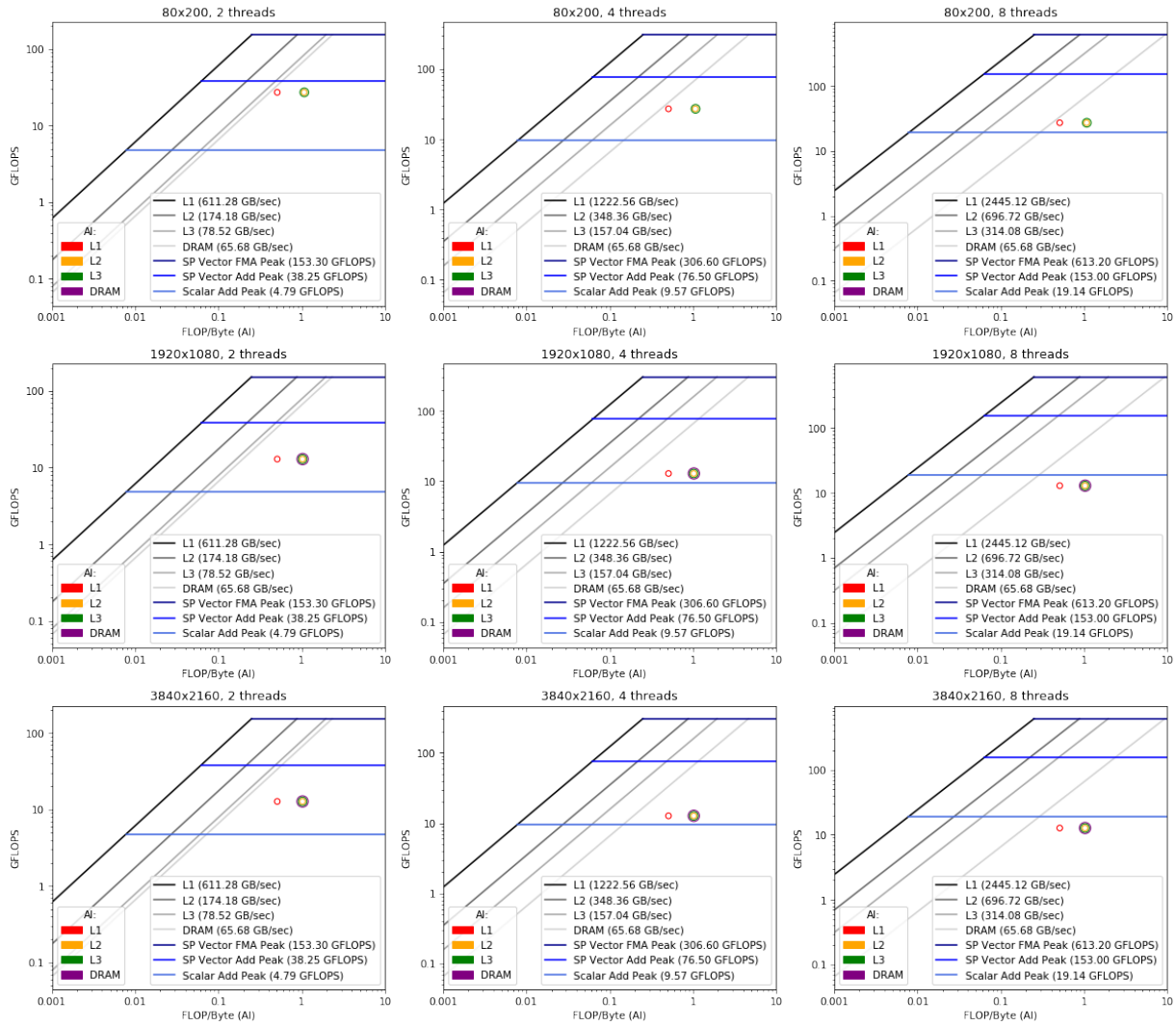


Figure 13: Parallel roofline prediction models using all values of T with the corrected AI's of the vectorized implementation.

References

[1] Cyrus K. Aidun and Jonathan R. Clausen. Lattice-boltzmann method for complex flows. *Annual Review of Fluid Mechanics*, 42:439–472, 2010. `doi:10.1146/annurev-fluid-121108-145519`.

[2] Tuomas Koskela, Zakhar Matveev, Charlene Yang, Adetokunbo Adedoyin, Roman Belenov, Philippe Thierry, Zhengji Zhao, Rahulkumar Gayatri, Hongzhang Shan, Leonid Oliker, et al. A novel multi-level integrated roofline model approach for performance characterization. In *International Conference on High Performance Computing*, pages 226–245. Springer, 2018.

[3] Timm Kruger, Halim Kusumaatmaja, Alexandr Kuzmin, Orest Shardt, Silva Goncalo, and Erlend M. Viggen. *The lattice boltzmann method, principles and practice*, volume 10. 2017. `arXiv:arXiv:1011.1669v3`.

[4] Dan Schroeder. Fluid dynamics simulation. URL: `https://physics.weber.edu/schroeder/fluids/`.