FAST CAVITY FLOW SIMULATION BASED ON THE NAVIER-STOKES EQUATION

Selin Barash, Peter Horcic, Filip Jakšić, Pavel Kliska

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

We optimize a cavity flow simulation which aims to simulate the motion of a liquid enclosed in a 2D cavity with a lid moving at a constant speed. We identify that the computations is memory bound and apply a so-called time skewing approach to improve cache friendliness. This results in a speedup of over 1600% in comparison with a straightforward baseline implementation.

1. INTRODUCTION

Computational fluid dynamics is a fundamental problem in hypersonics, meteorology and environmental engineering [1]. There is an indispensable need for fast simulations as typically these applications involve an iterative process with a lot of data that is read, written and moved around. Moreover, these simulations are used frequently in product development, and as a result, a lot of different variations need to be tested. Most of these simulations are based on the Navier-Stokes equation, a system of partial differential equations (PDEs), which admits no exact solution in its most general case. Therefore, in practice, numerical finite element methods are used to solve fluid dynamics problems. These methods boil down to solving the system of PDEs on a discretized mesh. In this project we tackled one such system of PDEs, namely the cavity flow problem which aims to simulate the motion of a liquid enclosed in a 2D cavity with a lid moving at a constant speed. For ease of implementation we discretize the cavity as an equally spaced 2D mesh.

At the core of solving a discretized Navier-Stokes equation on such a mesh is a *stencil computation* where the elements of the mesh at a particular time-step t are updated based on the values of its neighboring elements at previous time-step $t-1$. Stencil computations are notoriously cache unfriendly [2] because in order to update at time-step t, we need to access the updated neighbour values from $t-1$. This quickly introduces dependencies of stencil values on large neighbourhoods over a couple of time-steps which makes high performance difficult to achieve for simulation sizes that are beyond cache capacity.

Contribution. We introduce a fast implementation of the cavity flow simulation on a square 2D mesh based on the Navier-Stokes Equations that outperforms a straightforward baseline for large dimension sizes and achieve a speedup of 1600%. Moreover, we identify that the Clang compiler is particularly useful for this simulation as it performs optimizations such as vectorization in a more efficient way (see Section 4).

Related work. The bulk of the discretized Navier-Stokes equation on an equally spaced 2D mesh involves a stencil computation which has already been studied extensively. For example, as they are incredibly cache unfriendly, there exists a plethora of approaches to improve cache locality by tiling the iterative computation in a way that elements in cache are re-used as often as possible [2, 3, 4, 5, 6]. Moreover, one could apply the Fast Fourier Transform by viewing the stencil computation as an iterative matrix-vector product and then applying a DFT to solve it [7].

More specifically regarding cavity flow simulations, there is some work which aims to attain highly accurate and numerically stable solutions to the problem [8, 9].

2. BACKGROUND ON THE APPLICATION

In this section we briefly introduce the algorithm to compute our cavity flow simulation which is based on the Navier-Stokes equations.

We consider the 2-dimensional simulation of cavity flow on a $d \times d$ grid for $d \in \mathbb{N}$ (i.e. we restrict ourselves to square grid sizes). We will model the cavity flow simulation for $T \in \mathbb{N}$ time-steps. In each time-step and at every location in our $d \times d$ grid, we need to compute the two velocity components along the x-axis $\boldsymbol{u} \in \mathbb{R}^{d \times d}$ and y-axis $\boldsymbol{v} \in \mathbb{R}^{d \times d}$ as well as the pressure $\boldsymbol{p} \in \mathbb{R}^{d \times d}$ of the simulation. We remark that these 3 matrices depend on the current time-step $t \leq T$ that is being simulated. On the other hand, the fluid density ρ as well as the fluid viscosity ν are arbitrary constants throughout the simulation that determine the fluid that is being modelled. In this project, they will be set such that the simulation does not produce any over- or underflow.

The matrices $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{p}$ are governed by a system of PDEs, however, in our algorithm we will solve the discrete

analog of that system of PDEs, as presented in [10]. To that end, we begin with the initial condition that $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{p}$ are $\mathbf{0}$. Then, at each time-step $t+1 \leq T$ we update the matrix $\boldsymbol{u}$ as follows for all indices (i,j) that are not on the border of the grid

$$\begin{aligned} u_{i,j}^{t+1} = u_{i,j}^t &- u_{i,j}^t \frac{\Delta t}{\Delta x}\left(u_{i,j}^t\right. \\ &- u_{i-1,j}^t - v_{i,j}^t \frac{\Delta t}{\Delta y}\left(u_{i,j}^t - u_{i,j-1}^t\right) \\ &- \frac{\Delta t}{\rho 2 \Delta x}\left(p_{i+1,j}^t - p_{i-1,j}^t\right) \\ &+ \nu\left(\frac{\Delta t}{\Delta x^2}\left(u_{i+1,j}^t - 2u_{i,j}^t + u_{i-1,j}^t\right)\right. \\ &+ \frac{\Delta t}{\Delta y^2}\left(u_{i,j+1}^t - 2u_{i,j}^t + u_{i,j-1}^t\right), \end{aligned} \tag{1}$$

where $\Delta x = \Delta y = 2/(d-1)$ and $\Delta t = 1/1000$ (i.e. the fixed duration of the time-step). Similarly, we update $\boldsymbol{v}$ as such

$$\begin{aligned} v_{i,j}^{t+1} = v_{i,j}^t &- u_{i,j}^t \frac{\Delta t}{\Delta x}\left(v_{i,j}^t - v_{i-1,j}^t\right) \\ &- v_{i,j}^t \frac{\Delta t}{\Delta y}\left(v_{i,j}^t - v_{i,j-1}^t\right)) \\ &- \frac{\Delta t}{\rho 2 \Delta y}\left(p_{i,j+1}^t - p_{i,j-1}^t\right) \\ &+ \nu\left(\frac{\Delta t}{\Delta x^2}\left(v_{i+1,j}^t - 2v_{i,j}^t + v_{i-1,j}^t\right)\right. \\ &+ \frac{\Delta t}{\Delta y^2}\left(v_{i,j+1}^t - 2v_{i,j}^t + v_{i,j-1}^t\right), \end{aligned} \tag{2}$$

and finally, the pressure $\boldsymbol{p}$ is updated where we compute for T_p time-steps the following for every $t_p + 1 \leq T_p$ (the initial value for the pressure is the final value of the previous time-step t)

$$\begin{aligned} p_{i,j}^{t_p+1} = &\frac{\left(p_{i+1,j}^{t_p} + p_{i-1,j}^{t_p}\right)\Delta y^2 + \left(p_{i,j+1}^{t_p} + p_{i,j-1}^{t_p}\right)\Delta x^2}{2\left(\Delta x^2 + \Delta y^2\right)} \\ &- \frac{\rho \Delta x^2 \Delta y^2}{2\left(\Delta x^2 + \Delta y^2\right)} \\ &\times \left[\frac{1}{\Delta t}\left(\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}\right)\right. \\ &- \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}\frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} \\ &- 2\frac{u_{i,j+1} - u_{i,j-1}}{2\Delta y}\frac{v_{i+1,j} - v_{i-1,j}}{2\Delta x} \\ &- \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}\frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y}. \end{aligned} \tag{3}$$

At the grid edges, we set for all $j \in [d]$

$$\boldsymbol{u}_{d,j} = 1,\ \boldsymbol{u}_{j,d} = \boldsymbol{u}_{j,1} = \boldsymbol{u}_{1,j} = 0 \tag{4}$$

and for all $j \in [d]$

$$\boldsymbol{v}_{1,j} = \boldsymbol{v}_{j,1} = \boldsymbol{v}_{j,d} = \boldsymbol{v}_{d,j} = 0. \tag{5}$$

Finally, we demand that $\boldsymbol{p}$ must satisfy that

$$\partial_y \boldsymbol{p}_{i,0} = 0 \text{ for } i \in [d], \tag{6}$$

$$\boldsymbol{p}_{i,2} = 0 \text{ for } i \in [d], \tag{7}$$

$$\partial_x \boldsymbol{p}_{i,j} = 0 \text{ for } \{0,2\} \times [d]. \tag{8}$$

(We remark that even though we use the continuous notation, this is an abuse of notation and we still mean the discrete variables.)

Cost Analysis. We will count the number of floating point computations (flops) of the algorithm, specifically the number of additions, divisions and multiplications. This yields the cost estimate which depends on the dimension of the grid d

$$\begin{aligned} C(d) = &33400(d-2)^2 \cdot C_{\text{ADD}} + 11700(d-2)^2 \cdot C_{\text{MUL}} \\ &+ 74000(d-2)^2 \cdot C_{\text{DIV}}. \end{aligned} \tag{9}$$

The asymptotic runtime of the simulation is $\mathcal{O}(TT_pd^2)$ (here we did not consider T and T_p as constant).

3. OPTIMIZATION METHODS

In this section, we introduce each of the major optimizations we implemented.

3.1. Baseline

Our baseline is a line-by-line translation of the Python code of the cavity flow simulation [10]. It is significantly inefficient because it tries to mimic what the Python implementation does as closely as possible. For example, it allocates new arrays in the loops (via `malloc`) and frequently employs `memcpy` to copy arrays within loops like the Python implementation which further results in slower performance.

3.2. Standard Optimizations

In a first set of optimizations, we removed all unnecessary calls to `malloc` by preallocating the corresponding array before entering the loop. Moreover, we avoid using `memcpy` and instead simply swap the pointers using a temporary variable (i.e. strength reduction). We call this implementation *preallocated*. In a second step, we applied optimizations to simplify the arithmetic and reduce the flop count (for example, we eliminated common sub-expressions) by exploiting the fact that we only consider a square grid. We call this implementation *faster_math*. Thus, we managed to reduce

```
for n_p from 1 to T_p do
    swap p and pn
    for i from 2 to d-1 do
        for j from 2 to d-1 do
            p[i][j] = (pn[i][j+1]
            + pn[i+1][j] + pn[i][j-1]
            + pn[i-1][j] - b[i][j]) * 0.25
```

Fig. 1. Pseudocode of `pressure_poisson` after applying standard optimizations. We refer to the outer-most loop as the time loop and the 2 inner-most loops as the spatial loops.

the complexity of Equation (3) and the result of these optimizations is Figure 1. We call the outer-most loop the time loop and the inner most loop the spatial loops of Figure 1.

The profiler played a pivotal role in guiding us to determine which further optimizations to try. Importantly, we identified that `pressure_poisson` (see Figure 1) dominates 89% of the runtime of the entire computation, and hence, we will devote most of our attention to this. With the profiler, we identified that 67% of all cycles of the computation were spent on fetching data either from caches or DRAM, and 53% of cycles were solely due to accessing DRAM. Hence, our next attempt was to try a basic blocking mechanism to improve cache friendliness. To that end, we performed a basic blocking mechanism in the spatial loops of Figure 1 and tried different block sizes which were a lot smaller than 800 so that the entire computation would fit into the L1 cache. As the new loops from the blocking mechanism are constant (given a block size), we completely unrolled them (in order to improve the ILP) for a sufficiently small block size and performed common subexpression elimination. We call this implementation *blocking*. Unfortunately, we observed a performance reduction after attempting this optimization (see Section 4).

Further analysis revealed that we cannot attain any significant speedup from only blocking the spatial loops because they do not access the entire matrix, meaning that we do not reduce cache misses substantially enough (i.e. we only improve spatial but not temporal locality). The profiler reveals that we only have a 15% reduction in cycles spent on memory stalls. On the other hand, the blocking implementation executes three times more instructions than `faster_math` which outweighs the memory improvements resulting in a slightly lower performance than `faster_math`). We will return to this problem of cache locality in Subsection 3.4.

3.3. AVX2 Optimizations

In a next step, we tried to optimize the `pressure_poisson` more thoroughly with AVX2, specifically with packed double-precision `__m256d`. In a first step, we applied a basic vectorization technique without any data transformations to the layout of the array. This resulted in almost negligible speedup. Applying a more rigorous analysis with the profiler and inspecting the assembly code, reveals that the compiler was already vectorizing the entire function efficiently. Nevertheless, we attempted to out-smart the compiler by implementing more advanced data transformations as described in [11]. We call this implementation *aligned_avx*. This lead to similar results as before because after more careful consideration, we realized that the main benefit of this methodology was to avoid using unaligned loads (and instead use aligned loads), i.e. we transformed the array so that we would always have aligned data. This could be beneficial if the latency or throughput of aligned loads is better than that of unaligned. However, as unaligned and aligned loads have identical latency and throughput on Skylake, this yields no speedup. Moreover, according to [12], we should not worry about alignment in modern processors (i.e. newer than Skylake).

3.4. Advanced Optimizations

We will now describe our most advanced optimizations which resulted in the best speedup (see Section 4). As already stated in Subsection 3.2, the function in Figure 1 incurs an enormous number of cache misses. We realized that a better way to reduce cache misses was to not only block the spatial loops of the function (where each array element is only reused very infrequently), but instead focus on the time loop (where each array element is accessed multiple times).

Trapeze method. We implemented the algorithm described in [13] for a 2D stencil. For simplicity, we will first present how the algorithm works on an 1D stencil. Assume we want to compute some interval of values of the 1D stencil at time step t_1 and we know the values of the stencil computation at the time step t_0 on the interval $[x_0, x_1]$. Since every value depends on the value immediately to the left and to the right of it, we cannot compute the leftmost and rightmost value of the interval at time $t_0 + 1$, and the trapezoid narrows down as we move through time. This is visualised in Figure 2 where the x-axis represents the spatial dimension and the y-axis represents the time dimension. We can now represent our entire computation as an trapezoid which does not shrink because the border conditions only depend on the current iteration. The trapezoid representation of the problem enables us to do two kinds of cuts on the trapeze: *space cuts* and *time cuts*. If the width (w in Figure 2) of the trapezoid is at least twice the height, we perform a space cut which means that we split the trapezoid

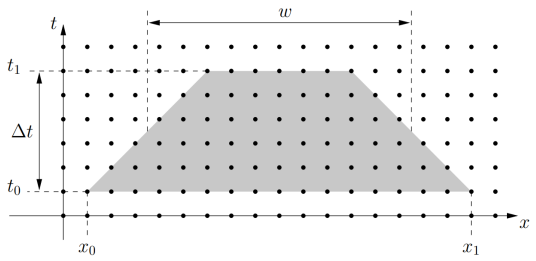

Fig. 2. Visualization of a trapezoid.[13]

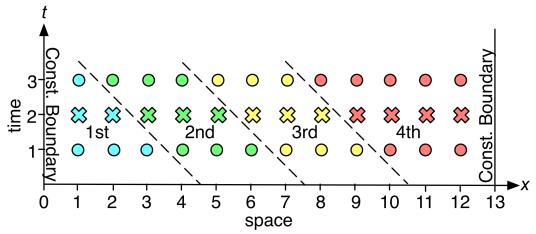

Fig. 3. Spacetime diagram of the time skewing approach for a three point 1D stencil.[2]

diagonally along its spatial dimension and compute its two halves recursively. Otherwise, we cut the trapezoid along its time dimension (i.e. a horizontally) and compute recursively. If the height is 1 (our base case), then we simply run the ordinary `pressure_poisson` on this trapezoid. Frigo et al. [13] claim that this approach has the asymptotically optimal cache hit rate despite it being cache-oblivious, meaning it does not need to know the cache size. To apply this method to a 2D stencil, we just need to be able to perform two kinds of *space cuts*. A *space cut* can now be applied if the "width" along that dimension is at least twice the height. We always first try to split along the x-dimension (rows of the array), then along the y-dimension (columns of the array), and finally along the time dimension. Since we run `pressure_poisson` T times in the simulation, an optimization opportunity is to precompute the recursion of this function, and memorizing for which trapezes we call the base case method. We call this implementation *precomputed_trapeze*. According to [2], further performance improvements can be attained by entering the base case early (i.e. not waiting until the height of the trapezoid is 1). This resulted in a new best performance for large d. Because we cut-off the recursion early, we call this implementation *cut-off_trapeze*. However, as the cut-off point depends on the cache-structure, this method is no longer cache-oblivious. Even though this was our last improvement with respect to the trapeze method, [2] proposes further improvements for this method, e.g. autotuning the cut-off point or not cutting across the y axis. However, as we were only achieving marginal performance gains with the trapeze approach, we decided to put this further optimizations on hold and moved on to the method we present in the subsequent paragraph.

Skewed method. We implemented the time skewing approach outlined in [2]. The idea is that we avoid refetching some elements of the array in every iteration of the time loop by moving to the next iteration of the time loop *before* the work in the current iteration of the time loop finishes. Or in the words of the trapeze method, we split the trapezoid into parallelepipeds (or skewed blocks) of the same size and compute the stencil parallelepiped by parallelepiped. The approach for a 1D three point stencil is illustrated in Figure 3: We partition the array into spatial blocks and on on a fixed block we iterate through time. This skews the block because of dependencies on other blocks. Once we computed all the values in the first block accounting for their dependencies, we move on to the next spatial block. The unfinished iteration of the time loop for the rightmost points will continue in the next block. Note that there is no dependency violation because the array that we are writing to is not the same as the array we are reading from. To apply this approach to a 2D stencil each of the 8 border conditions needs to be treated differently [1].

At first, we only considered square blocks, but as we realized that rectangular cache blocks may yield improved performance (as the y axis is contiguous in memory), we modified the implementation to work for rectangular blocks as well (reducing the number of 1-stride cuts is also mentioned in [2]). The ideal block sizes were found with an extensive autotuning approach where we examined over 900 different block size combinations (see Section 4.)

This was already sufficient to attain a new best performance, but as we were still not at peak performance, we tried to further improve it with two ideas. Firstly, we thought that we could further improve cache locality by explicitly optimizing for the L1 cache. To that end, we implemented more fine grained blocking along the space dimension by splitting the skewed blocks into skewed subblocks. We call this implementation *subskewed*. On the other hand, we noticed that the rectangular skewed implementation was back-end bound in the sense that it had sub-optimal port utilization (i.e. around 30% of the cycles of the entire computation only 0, 1 or 2 ports were being used). Thus, we tried to improve on the weak ILP by unrolling the inner-most loop of `pressure_poisson` and re-ordering the computations in a more ILP friendly way. We call this implementation *uskewed*.

Code generation. In the skewed method we first iterate over time-steps of a skewed block, then over rows and finally over columns. There is a lot of freedom in ordering these operations, and not all results need to be written to the memory because they get overwritten on subsequent time-step iterations. Reordering operations manually

[1] see implementation for full details

is a tedious task and would result in code that is potentially hard for the compiler to optimize. To provide the compilers with as many opportunities to optimize code as possible, we generated the completely unrolled single static assignment (SSA) code of computing 10 time steps on a 16×16 skewed block. We call this implementation *generated*. However this method did not yield expected improvements and was in fact slower (see Section 4).

4. EXPERIMENTAL RESULTS

In this section we present the setup of our infrastructure and analyze the performance of the optimizations of Section 3.

Experimental setup. We ran the experiments on an Intel Core i7 $-$ 6700. This is a Skylake processor with a base frequency of 3.4 GHz (with Turbo Boost disabled). The L1, L2 and L3 caches can store 32kB, 256kB and 8MB, respectively (where the L1 and L2 figures are per core). It can execute flops (including FMAs) in ports 0 and 1. Our main measurements were compiled with GCC 12.3.0 using the flags `-O3 -ffastmath -mavx2 -mfma`.

As input we considered the dimension $d \times d$ of our simulation (i.e. the size of our matrices $\boldsymbol{u}$ and $\boldsymbol{v}$). Moreover, all vectorized implementations must satisfy that $d \equiv_4 0$. The trapeze and skewed simulations must satisfy more involved constraints regarding the (sub-)block sizes which are verified with `assert` statements in the code (for brevity, we do not list them here). We kept the number of time-steps and the fluid density ρ as well as all other parameters constant. Some of these constants (e.g. ρ) were explicitly chosen so that the simulation would not cause under- or overflows.

In order to validate the correctness of our implementation, we check it against the reference Python implementation [10] by comparing the results on the 6 most significant digits. The runtime measurements were conducted with the CPU's Time-Step Counter `tsc_x86`.

Results of basic optimizations. In Figure 4, we see that our preallocated implementation where we avoid calls to `malloc` in every loop iteration reduces the runtime. However, our faster_math implementation, where we performed the other basic optimizations (i.e. simplifying the equations and precomputing the constants), reduced the flop count but did not improve the runtime. Evidently, GCC is already optimizing the arithmetic well. Nevertheless, we will compare to faster_math as it is more friendly to understand and more advanced optimizations can be implemented more easily. As already explained in Subsection 3.2, our blocking implementation fared worse than faster_math (see Figure 6).

Regarding AVX, we fully vectorized `pressure_poisson` and even applied the advanced data transformations described in Subsection 3.3, yet this resulted in no speedup. Inspecting the assembly code and profiler reveals that the compiler is already vectorizing this part of the code efficiently. This is

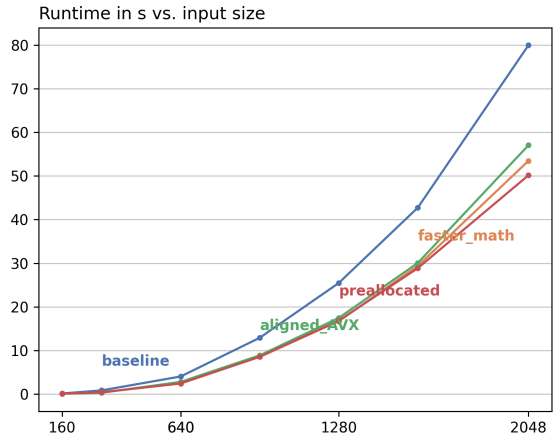

Fig. 4. Runtime of basic optimizations including AVX. Compiled with GCC.

perhaps unsurprising because the for loop is of a simple nature and the computations are straightforward. The reason why the data transformation did not help is that its main benefit is that it would use *aligned* loads instead of *unaligned* ones. However, on Skylake, both of these instructions have a latency of 7 cycles and a gap of 0.5 cycles per instruction, meaning that we have exactly the same runtime with or without transforming the array as can be seen in Figure 4.

Results of advanced optimizations. Firstly, we applied the recursive approach of the trapeze method. This resulted in worse performance in comparison to the standard optimization. We inspected the trapeze implementation with the profiler and noticed that 34% of cycles are spent on instructions concerned with control flow and function calls while it is only 2% for faster_math, suggesting that the deep recursion results in performance reduction. However, in Figure 5, we observe that its performance remains steady as it leaves the L3 cache at $d = 577$ (and the profiler does confirm that this method reduced the L3 cache miss rate from 44% to 7%), suggesting that it does help with cache performance. Thus, as there is an overhead from the recursion, we tried to improve on it by precomputing some of the values (i.e. dynamic programming) and terminating the recursion at an earlier point. Both of these greatly improve on the original trapeze implementation, and in fact, for some large d, we observe better performance in contrast to the standard optimization. Moreover, we similarly see steady performance when we deal with matrices greater than the L3 capacity. However, as we see in Figure 5, cutoff_trapeze performs again worse for $d > 2048$. This is likely because the precomputed data grows beyond L3 cache capacity, and hence, we no longer benefit from our precomputation. The

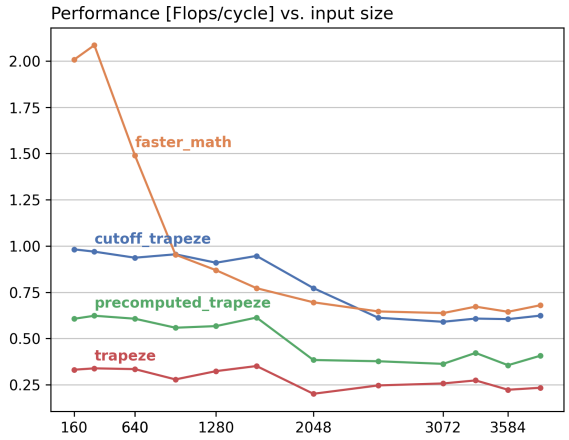

Fig. 5. Performance of trapeze methods. Compiled with GCC.

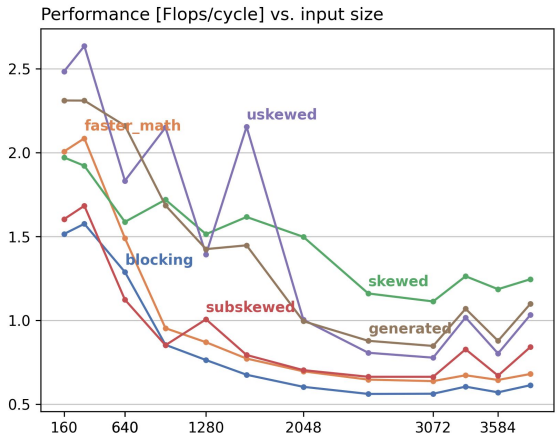

Fig. 6. Performance of blocking and skewed methods. Blocking was run with block size 4 and the blocking loop was completely unrolled. Compiled with GCC.

fact that precomputed_trapeze has a similar decrease in performance affirms this hypothesis. Thus, we need to consider a different approach that does not rely on recursion.

Indeed, the skewed method with rectangular blocks results in even better performance than any trapeze method. Comparing both implementations with the profiler, we see that skewed spends about 30% more cycles stalling for memory loads, but overall executes about half as many instructions concerned with control flow and function calls as trapeze. Hence, it appears that the main benefit of skewed is that it does not have a recursion overhead. Nevertheless, as can be seen in Figure 6, for $d \leq 577$ the skewed method is slower than the basic optimized version. Notice that for this upper bound of d, we have that $8 \times 3 \times d \times d \leq 8\text{MB}$, and hence, the entire data of `pressure_poisson` fits into the L3 cache, meaning that we do not benefit from our blocking. The fact that we are slower (instead of just as fast) could be due to the overhead of the skewing, i.e. stalling the time loop and returning to it later reduces the compiler's ability to optimize it.

We determined the optimal block size using an autotuning approach. We exhaustively searched over 900 different combinations for the block size, repeating the same search over different matrix sizes. There are different block sizes that attain the optimal performance. They all share the property that they are longer along the y-dimension than along the x-dimension and that they comfortably fit into the L2 cache. This is unsurprising because the array is contiguous in memory along the y-axis (so we benefit from stride access to reduce the cache miss rate), and we obviously want our data to fit into the L2 cache so we can benefit from the high bandwidth rate from the L2 cache. Notice that we do not optimize for the L1 cache as it is too small to hold a large enough matrix to reduce the cache miss rate substantially. However, as claimed in [2], the optimal parameters for the block sizes remain unintuitive. For example, it is not clear why the optimal block size does not perfectly fit into the L2 cache (i.e. with the optimal block size the L2 cache is only up to 30% saturated).

The subskewed implementation, our refinement of skewed where we tried to improve L1 cache locality, resulted in worse performance (see Figure 6). An analysis with the profiler reveals subskewed reduced L1 cache misses by 63%, but also increased the number of retired load instructions by 25%, and and thus spends 22% more cycles stalling for memory loads (in comparison to skewed). Our other refinement of the skewed method, meaning uskewed with loop unrolling and computation reordering, managed to out-perform skewed for some dimension sizes: $d \in \{160, 320, 960, 1600\}$. The profiler reveals that uskewed managed to improve the port usage (as intended) by over 30% for small d. However, when d becomes too large, uskewed becomes 100% more memory bound than skewed, specifically the L3 latency is the bottleneck, and this causes a performance decrease. Evidently, the better port usage is not enough to compensate the worse memory usage, and overall, skewed remains superior in the average case. However, for small d, uskewed does benefit from the improved port usage and outperforms skewed.

We tried vectorizing the skewed implementation manually (preliminary measurements indicated no improvement), and as the profiler reported that it was already fully vectorized and previous vectorization optimizations yielded no improvements, we did not consider this further.

Code generation for skewed proved to be unsuccessful

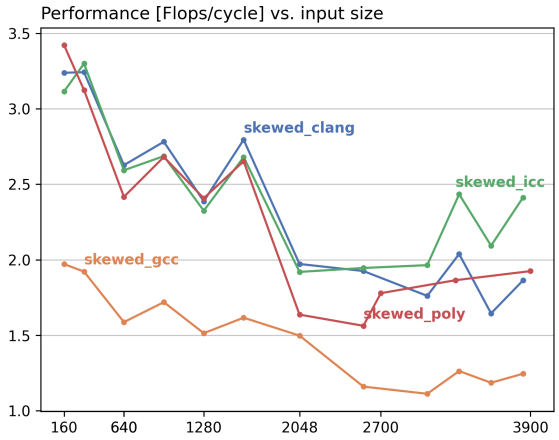

Fig. 7. Performance comparison of skewed with different compilers.

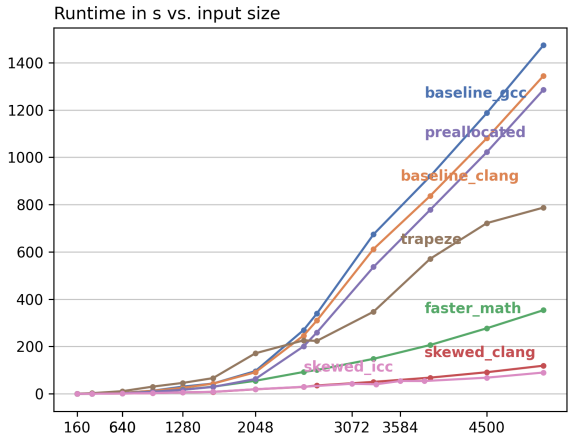

Fig. 8. Runtime of various methods. All of them were compiled with Clang unless noted otherwise.

for large d but it is at least as good as skewed for $d \leq 756$. The profiler shows that code generated spend 6.5x more cycles stalling due to memory than regular skewed. The bottleneck appears to be the DRAM bandwidth which suggests that our generated code cannot efficiently handle dependencies and the compiler cannot help with this (perhaps due to the fact that the generated function is over 60k lines long). This is not a problem for small d as we are in the L3 cache anyways. Perhaps this could be improved by generating smaller code (e.g. code-generating one time-step of a skewed block and not unrolling the time loop completely). However, as it was worse than skewed and its compilation time was too long, we did not try this further.

Different compilers. In addition to standard GCC, we tried to see if other compilers could produce a greater speedup. Specifically, we tried ICC 2024.1.2.20240508 and Clang 14.0.0. Then, we tried Polly 14.0.0 which is a high-level loop and data-locality optimizer [14]. Compiling with both Clang, ICC and Polly resulted in improved performance compared to GCC. For our largest matrix size, the performance gains range from 1.4x to 2x as can be seen in Figure 7. The greatest gains were with ICC. Also notable is that Polly did not manage to optimize our implementations better than plain Clang, which shows that even state-of-the-art optimization tools do not succeed with further optimizations.

In Figure 8, we show our final result of important optimizations with our optimized skewed implementation (i.e. with autotuning and some loop unrolling where applicable). This resulted in a final speedup of more than 1600% for $d = 5100$ when compiled with ICC.

5. CONCLUSIONS

In conclusion, we optimized the cavity flow simulation and obtained a speedup of over 1600% in comparison to a straightforward baseline optimization. The most successful optimization was a sophisticated time skewing approach that improved cache locality. This was further improved by performing autotuning on over 900 combinations of the block size of the skewing and some loop unrolling and compiling the code with Clang. Further avenues of optimizations could be to improve the code generation to generate shorter code (while also trying to optimize the dependencies of the calculations) or combining uskewed and subskewed as they have complementary problems (i.e. uskewed performs well with respect to ILP but has worse memory usage).

6. CONTRIBUTIONS OF TEAM MEMBERS

Selin. Implemented the initial version of the `blocking` method and experimented with different block sizes. Implemented the auto-tuning for `skewed` and `uskewed` methods to find the optimal block sizes for x and y dimensions.

Peter. Fixed some bugs in `faster_math` and tried to improve on it with `fasterfaster_math`. Implemented loop unrolling with scalar replacement and common subexpression elimination in the `pressure_poisson` function. Otherwise, mainly focused on SIMD optimizations: started with a `vanilla_avx` implementation to see that the compiler is indeed vectorizing properly, then tried to improve on that by implementing data transformations as described in [11]. Analyzed the basic `skewed` implementation with the profiler and implemented loop unrolling for the most critical section of `skewed` (resulted in some speedup).

Tried to apply AVX to skewed. Wrote the report.

Filip. Analyzed all implementations with the profiler and provided data and explanations for bottlenecks and further optimization directions. Wrote and managed the infrastructure for experiments.

Pavel. Implemented the initial versions of the following methods: `baseline`, `preallocated`, `faster_math`, `aligned_AVX`, `trapeze`, `precomputed_trapeze`, `cutoff_trapeze`, `skewed`, `subskewed` and `uskewed`. Looked at the previous research on efficient stencil computation and found papers describing the trapeze and time-skewing approaches to make stencil computations more cache-friendly. Proofread the final report.

7. REFERENCES

[1] Jiri Blazek, *Computational fluid dynamics: principles and applications*, Butterworth-Heinemann, 2015.

[2] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 workshop on Memory system performance and correctness*, 2006, pp. 51–60.

[3] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel, "Cache oblivious parallelograms in iterative stencil computations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, New York, NY, USA, 2010, ICS '10, p. 49–59, Association for Computing Machinery.

[4] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel, "Cache accurate time skewing in iterative stencil computations," in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 571–581.

[5] Alain Denzler, Geraldo F Oliveira, Nastaran Hajinazar, Rahul Bera, Gagandeep Singh, Juan Gómez-Luna, and Onur Mutlu, "Casper: Accelerating stencil computations using near-cache processing," *IEEE Access*, vol. 11, pp. 22136–22154, 2023.

[6] Thomas L Falch and Anne C Elster, "Register caching for stencil computations on gpus," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2014, pp. 479–486.

[7] Zafar Ahmad, Rezaul Chowdhury, Rathish Das, Pramod Ganapathi, Aaron Gregory, and Yimin Zhu, "Fast stencil computations using fast fourier transforms," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 8–21.

[8] Carlos Henrique Marchi, Roberta Suero, and Luciano Kiyoshi Araki, "The lid-driven square cavity flow: numerical solution with a 1024 x 1024 grid," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 31, pp. 186–198, 2009.

[9] Yiannis Papadopoulos, "A driven cavity exploration," `https://www.acenumerics.com/the-cavity-sessions.html`, June 2015.

[10] Lorena A. Barba, "12 steps to navier–stokes," `https://nbviewer.org/github/barbagroup/CFDPython/blob/master/lessons/14_Step_11.ipynb`, 2017.

[11] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*. Springer, 2011, pp. 225–245.

[12] Intel, "Intel 64 and ia-32 architectures optimization reference manual," April 2012.

[13] Matteo Frigo and Volker Strumpen, "Cache oblivious stencil computations," in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, pp. 361–366.

[14] Tobias Grosser, Armin Groesslinger, and Christian Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, pp. 1250010, 2012.