Optimizations of Jos Stam's Fluid Algorithm

Parallel Computing - Phase III

Carlos Ribeiro
PG55926

Diogo Matos
PG55934

Júlio Pinto
PG57883

***Abstract*—This report explores the optimizations made to Jos Stam's Fluid Algorithm for calculating real-time fluid dynamics. The optimized code was written in C++. This report outlines the optimization of this algorithm through three distinct phases. In the first phase, execution time was enhanced by implementing various code-level improvements for a single thread. The second phase introduced shared memory parallelism utilizing OpenMP to achieve strong scalability. Finally, the third phase leveraged GPU parallel programming with CUDA, further boosting performance and showcasing the advantages of heterogeneous computing.**

***Index Terms*—Fluid dynamics, Parallel computing, Algorithm optimization, CUDA, OpenMP, GPU acceleration, Gauss-Seidel method**

I. Introduction

The primary objective of this project is to investigate and assess various techniques and tools designed to optimize an algorithm's performance. This endeavor allowed us to examine multiple parallelization models to enhance our algorithm's efficiency.

In this report, we discuss all phases of the project, the decisions behind our choices, and the results obtained from each approach. It is important to note that the code was modified between the first and second phases; however, certain observations remained consistent across both versions.

We outline the project's development through its different stages, beginning with the initial creation of the algorithm in its sequential form, followed by a series of incremental optimizations. Next, we introduce shared memory parallelism using OpenMP to achieve strong scalability, and finally, we integrate the CUDA programming model to leverage GPU acceleration, demonstrating the benefits of heterogeneous computing.

II. Identifying Hot Spots

After running profilers and analyzing the code deeply in both versions, we understood that **the biggest hot spot was the function `lin_solve`**. In the first phase, this function had a data dependency, wherein the current iteration depended on the previous one. This was resolved in phase two, using the red-black method to remove said data dependency and allow for vectorization and parallelization. Nonetheless, even with this change, `lin_solve`, remained the biggest hot spot in the code provided for both versions.

Smaller functions, like `set_bnd`, despite being called a lot of times don't require as much as `lin_solve` in terms of computational resources.

III. Single-Thread Optimizations

We wound up applying several single-thread optimizations, the most relevant of which being those that affected `lin-solve`.

A. *Compiler Optimizations*

To begin with, we had to use the best flags possible to optimize our time: `-Ofast -march=native -ftree-vectorize`

These flags allowed us to focus more on the code provided and smaller nuances that the compiler doesn't automatically optimize.

B. *Optimization of Divisions*

To begin with, we identified all instances where divisions were being performed. We noticed that many of these divisions occurred repeatedly within loops, especially in `lin_solve`. Since the denominator values remained constant throughout these loop iterations, we optimized the process by precomputing the inverse of the denominator and storing it in a variable before entering the loop. This way, we replaced the division operations inside the loop with multiplications, which are generally 3 to 5 times faster. This optimization was applied across all functions with multiple iterations to improve overall performance.

C. *Spatial Locality*

In the `IX(i, j, k)` macro, we observed that the array was organized with k as the outermost dimension and i as the innermost. However, upon analyzing the loop structure, we found that the array was being accessed with i as the outermost loop and k as the innermost. This discrepancy adversely affects spatial locality, causing the program to jump around the array instead of accessing it sequentially.

Initially, the array was accessed in the following order:

`[1, 4, 7, 10, 2, 5, 8, 11, 3, 6, 9, 12]`

By modifying the loop iterations, we changed the access pattern to:

`[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]`

This adjustment ensured that the array elements were read and written in a sequential manner. As a result, we improved spatial locality, which enhanced cache performance and led to better overall execution times.

D. *Memory Tiling*

To handle the large datasets processed by our loops, we refactored them to iterate in blocks, aiming to optimize cache usage. We employed a gradient-based approach to determine the optimal block size and identified that a block size of 4

(among powers of 2) yielded the best results. This change significantly improved performance by reducing the frequency of cache misses.

E. Vectorization

In the initial phase of our project, we identified that the `lin_solve` function could not be fully vectorized due to data dependencies (iteration dependencies) within its loop. To address this limitation, we implemented a pre-processing loop that calculates certain values required by the dependent loop. This approach was feasible because these pre-computed values were only used for multiplication or addition operations with the dependent variables. By separating the computation of these values, the dependent loops could be vectorized either manually or by the compiler. This modification enhanced the vectorization potential of the `lin_solve` function, resulting in a small improvement in performance.

F. Results

As specified by the assignment, multiple tests took place on the SEARCH cluster using gcc 11.2.0 in the partition `cpar`. With that in mind, we obtained the following time (`SIZE` = 42):

$$3.1624 \text{ seconds} \tag{1}$$

Maintaining the same value of 81981.6 and with a total of 2.95% misses of all L1-cache hits.

IV. Multi-Thread Optimizations (OpenMP)

When developing the Multi-Thread version of our program the code given was changed, this way the data dependency from the previous `lin_solve` was removed.

To enhance the performance of the code, we employed OpenMP pragmas to parallelize the computational hot spots, focusing on distributing workloads efficiently across available CPU cores. The following approaches were implemented for different parts of the code:

A. General Loop Parallelization:

The **`#pragma omp parallel for`** directive was utilized for all independent `for` loops, specifically those that could be effectively parallelized. This enabled the iterations to be distributed across multiple threads, ensuring balanced workload allocation and improved execution speed.

The loops within the **`add_source()`, `advect()`, and `project()`** functions exclusively employed `#pragma omp parallel for` due to the **absence of dependencies between iterations**. Additional clauses, such as `private` or `reduction`, were deemed unnecessary, as these functions **did not require private variables or the aggregation of values** across iterations.

Overall, the `collapse` clause wasn't utilized throughout the code, as its application did not produce sufficiently improved results to justify the associated **increase in overhead**.

B. Optimizations in `lin_solve()`:

The `lin_solve()` function, the most computationally intensive part of the algorithm, required special handling for parallelization. This function **performs multiple nested loops** while solving linear equations iteratively using the Gauss-Seidel method. To **correctly compute the maximum value of `change`**, ensuring an accurate convergence, while maintaining thread safety, we used the **`reduction` clause**. The `reduction(max:max_c)` clause aggregates the maximum difference (`max_c`) across threads after each iteration, **preventing race conditions** while monitoring convergence. It also **eliminates the need for explicit synchronization**, like critical sections, **reducing overhead and improving efficiency** of the parallel code.

We also **used `private(old_x, change)`** to guarantee that each thread had its own set of `old_x` and `change` variables, **not allowing data races** to occur, since these variables are accessed in every iteration of the loop and would otherwise try to be accessed by different threads, leading to a race condition.

C. Analysis of the implementation

One thing we realized after submitting phase 2 is that we misunderstood some documentation, this way for this phase we corrected the concurrent accesses we could find. **Notably the concurrent access in the function `set_bnd`, by putting the innermost loop variable `i`, inside a `private` clause.**

D. Results

As specified by the assignment, multiple tests took place on the SEARCH cluster using gcc 11.2.0 in the partition `cpar`. For these tests, we utilized 20 threads. With that in mind, we obtained the following time (`SIZE` = 84):

$$2.090s \text{ seconds} \tag{2}$$

Achieving the value of 140880.

V. CUDA Optimizations

In this phase, we transitioned our existing codebase to leverage the CUDA Library for GPU programming, aiming to harness the parallel processing capabilities of modern GPUs to accelerate our computations.

A. Analyzing the code

Our solver employs multiple nested loops to perform iterative computations. By utilizing CUDA, **we can distribute these loop iterations across multiple threads**, enabling concurrent execution and significantly reducing computation time. To achieve optimal parallelism, **we decided to create a dedicated kernel for each loop present in the code**. This approach allows us to efficiently distribute the workload across the GPU's threads, ensuring that each loop's iterations are processed simultaneously without bottlenecks.

B. General Optimizations

To maintain code clarity and efficiency, we adopted a strategy of **allocating only essential variables and data structures on the GPU device**. By minimizing the data transferred between the host (CPU) and the device (GPU), we reduce the overhead associated with memory transfers. Specifically, variables are initialized on the device, processed entirely

on the GPU, and then transferred back to the host only after computations are complete. This approach minimizes the number of necessary memory copies, enhancing overall performance.

For the majority of our kernels, we employed a **three-dimensional grid configuration** with blocks sized in a **32×8×1 pattern**. This configuration was chosen for several reasons:

- **Alignment with GPU Architecture:** The block size aligns with the GPU's warp size of 32 threads, ensuring that each warp is fully utilized and reducing thread divergence.
- **Memory Coalescing:** The 32×8×1 layout promotes efficient memory access patterns, allowing threads to access contiguous memory addresses and maximizing memory throughput.
- **Performance Testing:** We experimented with alternative block sizes, such as 8×8×8, but found that these configurations did not match the performance gains achieved with the 32×8×1 pattern. Extensive benchmarking confirmed that the chosen block size **enhances memory throughput and maximizes the GPU's computational performance**, making it particularly well-suited for the diverse operations within our fluid solver.

Further details on how different block configurations impact our solver's performance are discussed in subsequent sections of this report.

a) ***`add_source`:*** The `add_source` function is designed for simplicity. We implemented it using a **one-dimensional grid layout**, ensuring that each thread accurately writes to its corresponding position in the array. This straightforward approach minimizes complexity while maximizing performance, allowing the GPU to handle data additions seamlessly without unnecessary overhead.

b) ***`set_bnd`:*** The `set_bnd` function manages various boundary conditions for different variables, each requiring unique handling. To optimize performance and minimize thread divergence, we **decomposed this functionality into four distinct kernels**. By isolating each boundary condition into its kernel, we ensure that each kernel operates with a uniform execution path, eliminating unnecessary branching within threads. Although this strategy introduces additional overhead due to multiple CUDA API calls to `cudaLaunchKernel`, our observations indicate that it **reduces latency and enhances overall performance**, compared to previous iterations of this function.

c) ***`diffuse`:*** The `diffuse` function remains largely unchanged in its core logic. The primary modification involves **receiving GPU-specific variables**, enabling the function to operate directly on data stored in GPU memory.

d) ***`advect`:*** The `advect` function is responsible for transporting quantities such as velocity and density within the fluid simulation. Similar to `add_source`, this function is inherently straightforward. However, we enhanced its performance by employing a **three-dimensional grid layout** for its kernel, due to its nature. This configuration aligns seamlessly with the simulation's spatial dimensions, ensuring **efficient memory access** and optimal utilization of GPU resources. By structuring the grid in three dimensions, we facilitate better data locality and parallelism, which are crucial for the high-performance requirements of fluid dynamics simulations.

e) ***`project`:***

The `project` function maps the three-dimensional simulation space onto the GPU using a **block size of (32, 8, 1)**. This specific configuration is meticulously chosen to **fully leverage the GPU's parallel processing capabilities** while maintaining optimal memory access patterns. By selecting these dimensions, we achieve a **balanced distribution of workload** across CUDA blocks, ensuring that each block operates efficiently without overloading the GPU. This balance maximizes the performance of both the `project` kernel and the overall simulation, facilitating smooth and rapid computations within the fluid solver.

f) ***`lin_solve`:*** The `lin_solve` function underwent extensive experimentation and optimization, warranting a dedicated chapter in our report. Through rigorous testing and iterative refinement, we developed a highly efficient implementation tailored to our solver's specific needs.

VI. Implementing `lin_solve`

This function is without a doubt the most time-consuming in our whole project. Knowing that we tried to minimize the time spent on it and all the calls it does.

A. *Implementing `kernels`*

To efficiently implement the **red-black Gauss-Seidel method**, we **divided the computation into two separate CUDA kernels**. Each kernel is responsible for updating either the black or white cells of the grid.

As previously mentioned, these kernels utilize a **thread block configuration of (32, 8, 1)**. Unlike other kernels, the `lin_solve` kernels **adjust the grid dimensions** to account for the fact that each kernel processes only half of the grid. Specifically, the **`x` dimension is divided by two** to align with the red-black partitioning. This adjustment ensures that each thread block is assigned to a **distinct subset of the grid**, thereby reducing the computational load per kernel launch and enhancing overall performance.

To achieve this, the **`i` index is incremented by a stride of two**, ensuring that each thread operates on an alternating grid cell. The offset applied to the `i` index is determined by whether the kernel is handling black or white cells, ensuring correct alignment with the grid's parity.

During each iteration of the linear solver, the `lin_solve_black` and `lin_solve_white` kernels are **launched sequentially**. This alternating execution pattern adheres to the red-black ordering, ensuring that **dependencies between grid points are properly managed**. By processing all black cells followed by all white cells, we maintain the integrity of the Gauss-Seidel updates while fully leveraging the GPU's parallel capabilities.

B. *`lin_solve` convergence strategy*

`lin_solve` should stop when either the maximum number of iterations is reached, or the maximum change in any given value of the tensor* is below the tolerance 1e-7, this second

condition is what we call the **convergence criterion** and it can be expressed as $\max_c < \text{tol}$ which is equivalent to $\forall c \in \mathbb{C}. c < \text{tol}$ with $\mathbb{C}$ defined as the se of all changes. Also

$$\begin{aligned} \text{converged } C &:= \forall_{c \in C}.\ c < \text{tol} \\ \text{converged } C &\Leftrightarrow \neg\exists_{c \in C}.\ c \geq \text{tol} \end{aligned} \tag{3}$$

This formalization allows us to test for convergence without having to store a tensor with all the changes and somehow find its maximum.

$$\begin{aligned} &\text{let } \{C_1, C_2 ... C_n\} \text{ be a partition of } C \\ &\text{converged } C \Leftrightarrow \forall_{i \leq n}.\ \text{converged } C_i \Leftrightarrow \\ &\text{converged } C \Leftrightarrow \neg\exists_{i \leq n}.\ \neg\text{converged } C_i \end{aligned} \tag{4}$$

The idea of partitions lends itself nicely to how processing is done in the GPU with CUDA. Each partition represents a block of threads.

If we want to test if something is true across all threads, we can start by creating a shared boolean variable set to true and, if any thread finds a counter-example it sets it to false.

This attribution doesn't need to protect itself from data races, since the variable starts as true and any thread that finds a counter-example, sets it to false, meaning that the boolean (after initialization) only ever gets set to false (this action is independent of the state of the variable).

We tested two approaches:

1) Simply a global variable that may be edited by any thread.
2) One shared variable in each block that may be edited by any thread in said block and, when the kernel ends, the 0th thread edits a global variable

C. `lin_solve` convergence optimization

The following tests were done with a size of 84 because some aspects were tested in the sequential version, as well as, on the partition `day` due to the availability of the cluster.

This section aims to discuss the advantages and drawbacks of the optimization explained in the previous section.

Let's consider three approaches:

0) No convergence
1) Convergence with global variable (only)
2) Convergence with shared variable (+ global variable)

Our implementation of approaches 1 and 2 necessitates the use of `cudaMemCpy` and `cudaMemSet`, furthermore, the implementation of approach 2 entails the usage of two parallel barriers (upon setting a shared `local_done` variable, and upon writing it to `global_done`)

Thus, there is a tradeoff between the number of iterations of `lin_solve` and the cost of each iteration and workload symmetry.

In light of this, to get an idea of the average of wasted iterations, we analyzed the number of necessary iterations to converge, and these were the results:

`1:20.5%, 3:1.6%, 4:33.8%, 5:7.8%, 6:8.5%, 7:2.3%, did not converge:25.3%`

making the weighted average a little less than 8 iterations.

This strikes us as a significant gain (2.5) but only if each iteration is less than 2.5 times slower than in approach 0.

Upon implementation and testing, we reached the following table.

In this test, the gain in iterations is then 2.23, not far off from the estimate above.

approach	total	calls	avg	min	max
0	871ms	12k	72.6µs	71.8µs	83.7µs
1	437ms	5380	81.2µs	80.5µs	92.2µs
2	481ms	5380	89.3µs	87.2µs	92.1µs

a) ***Workload:***

`MemCpy` of the boolean variable takes 42 µs, `MemSet` takes 4 µs

- Approach 0 kernel: takes about 73 µs
- Approach 1 kernel: takes about 81 µs (only 8 more!)
  - workload: 81 + 46 = 127 µs
- Approach 2 kernel: takes about 89 µs
  - workload: 89 + 46 = 135 µs

b) ***Workload symmetry:***

- Approach 0: the workload is theoretically symmetrical between threads, but exhibits a range of 12µs, however heavily skewed to the lower end.
- Approach 1: each thread does some setup, some reads from global memory and one writes to global memory, or two, in the case where it's found a counter-example to the conversion criterion, also exhibits a range of 12µs, heavily skewed to the lower end.
- Approach 2: Does all that approach 1 does, however, the synchronization step is done by only one thread, and we also need to sync all threads upon initialization of the local variable, increasing the imbalance in workload …

c) ***Implemented Approach:***

Approach 1 "wins" – We can somewhat safely assume that the extra 9 µs in the approach 1 kernel are due to the conditional global memory write, even when adding to it the cost of `memCpy` and `memSet` stays below 2.23 times the time of approach 0.

This result initially went against our intuition that having a large number of threads writing to global memory would be slower than writing to a shared variable, and then having one thread per block writing it to global memory.

This happens because writes to global memory are buffered at the warp level, and the threads running the code don't "wait" for the write to conclude.

D. Results

As specified by the assignment, multiple tests took place on the SEARCH cluster using gcc 7.2.0 and cuda 11.3.1, in the partition `cpar`. With that in mind, we obtained the following time (`SIZE` = 168):

$$11.418 \text{ seconds} \tag{5}$$

Achieving the value of 160835.

VII. Testing

To effectively compare each version of our code, we conducted multiple tests across different configurations.

In this chapter, we present the results and the methodology used to gather them. All benchmarks for the single and multi-threaded versions were performed on the `day` partition using `c20` machines, while the CUDA versions were tested on `k20` machines.

A. Single-Thread Results

We extensively tested our single-threaded code with various problem sizes to evaluate its overall performance. The results, depicted in the accompanying graph (Fig. 4), demonstrate that as **the problem size increases, the execution time grows significantly**. This behavior is expected because most of **our code operates with a complexity of $O(N^2)$ or $O(N^3)$**, directly impacting performance as the problem scales.

B. Multi-Thread Results

a) ***Strong Scalability:*** Our performance analysis of the OpenMP implementation shows that with a small number of threads, we achieve near-linear speedup, indicating effective workload distribution. However, **as the number of threads increases, the speedup gains diminish** due to factors such as synchronization overhead and resource contention. This trend is illustrated in the speedup graph (Fig. 1), which shows rapid initial improvements followed by a plateau and eventual decline. Additionally, the efficiency graph (Fig. 3) reveals that efficiency remains high with fewer threads but drops sharply as more threads are utilized.

b) ***Overall analysis:*** After thorough analysis, we chose to use **20 threads** as the default setting for the multi-threaded version in our future tests, balancing performance and resource utilization.

Much like its single-thread counterpart, this version's runtime also increases as the size of the problem grows. (Fig. 4)

C. CUDA Results

For our GPU implementation, we conducted extensive tests to validate our approach.

a) ***Strong Scalability:*** Through careful evaluation of various block configurations, we found that the **(32, 8, 1)** block pattern delivers optimal performance for our implementation, as shown in the corresponding graph (Fig. 5).

b) ***Overall analysis:*** We also examined how our CUDA-based solution scales with larger data sizes. The results indicate that the overhead introduced by the CUDA API makes it **less effective for smaller datasets**. However, as the problem size increases, this overhead becomes negligible compared to the performance gains, demonstrating that **CUDA is increasingly beneficial for larger-scale problems** (Fig. 4).

VIII. Future Work

A. `lin_solve` GPU

`lin_solve` was implemented with two kernels, it may be possible to have a synchronization step and fuse the two, and although this could be costly, it may be the case that it's less costly than spawning a kernel.

Also, the threads in either kernel exhibit a stride of 2, meaning that adjacent threads in a block read and write to positions that are not adjacent, this results in sub-optimal spatial locality, a simple way of resolving this issue would be to split the array in two, in the host, before calling the kernels. We'd, however, have to analyze whether the overhead of said splitting operation would be compensated by the speedup in the kernels.

B. Combining kernels

Even though we separated `set_bnd` kernels we believe there might be a way to optimize it by combining. `set_bnd` is one of the most called functions in the solver. By reducing the the number of kernels it employs and somehow also guaranteeing that there is no thread divergence we could obtain 1/4 of the kernel calls for that function.

C. CUDA Streams

For future work, we plan to implement **CUDA streams** to achieve enhanced parallelism. Many functions currently iterate over different arrays sequentially. By utilizing CUDA streams, we could execute operations such as the three `add_source` calls in `vel_step` independently. Since these calls operate on separate arrays and have no dependencies between them, CUDA streams would allow them to run concurrently, potentially increasing overall performance, even with the additional overhead of creating and utilizing them.

IX. Conclusion

Throughout the semester, the three assignments guided us in learning and applying optimization techniques for both CPU-based and GPU-based parallel computing environments. In the first assignment, we enhanced the execution time of the original sequential code by implementing key optimizations such as algorithmic improvements, **instruction-level parallelism (ILP)** enhancements, and better memory organization to reduce cache misses. These changes led to significant performance gains and gave us a strong foundation in low-level optimization strategies.

In the second assignment, we introduced **OpenMP** to parallelize the optimized code, effectively utilizing multi-core CPU architectures. This allowed us to achieve further reductions in execution time by focusing on workload distribution, synchronization, and minimizing thread contention. Finally, in the third assignment, we explored the **CUDA programming model** to implement GPU-based parallelism. Although CUDA did not outperform OpenMP for smaller problem sizes, it showed superior performance compared to the sequential version for more complex problems. This experience deepened our understanding of GPU architectures and parallel programming, preparing us to optimize software for diverse hardware platforms.

X. Appendix

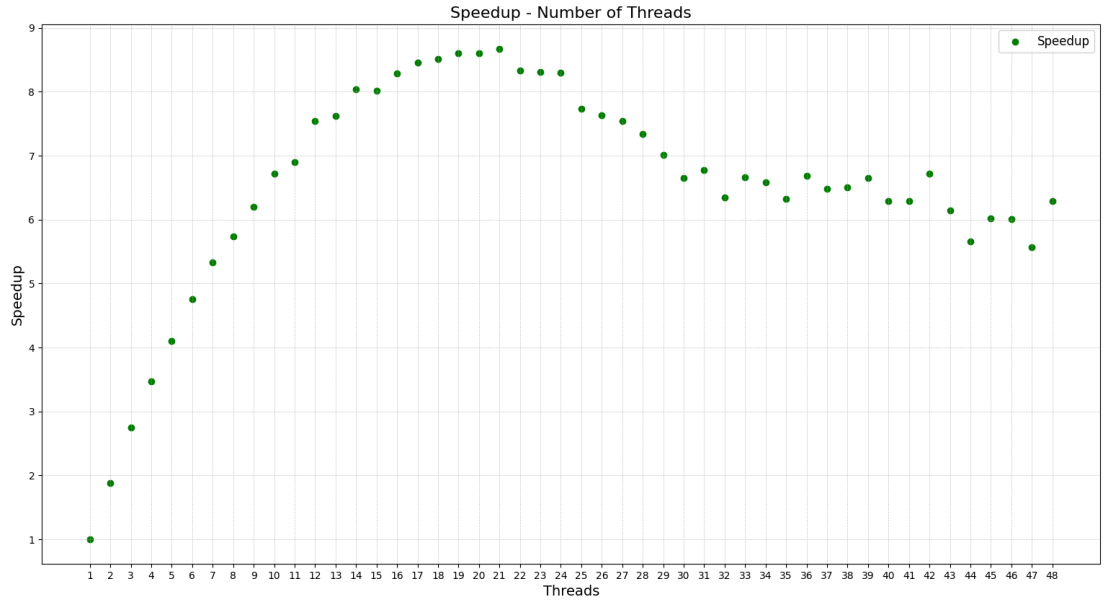

Fig. 1: Speedup Graph (partition `day`)

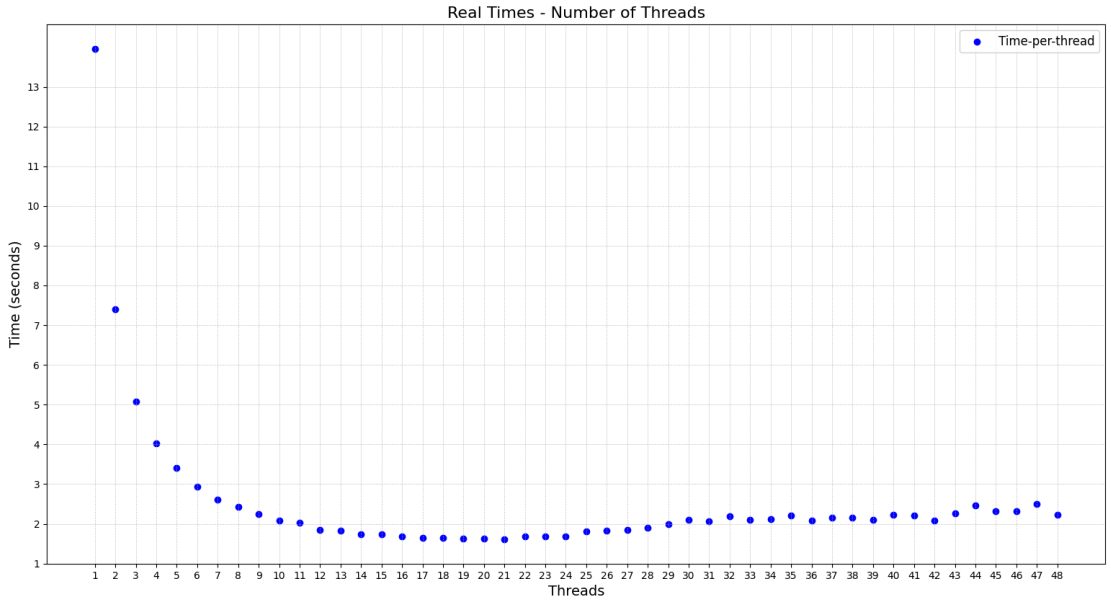

Fig. 2: Runtime Graph (partition `day`)

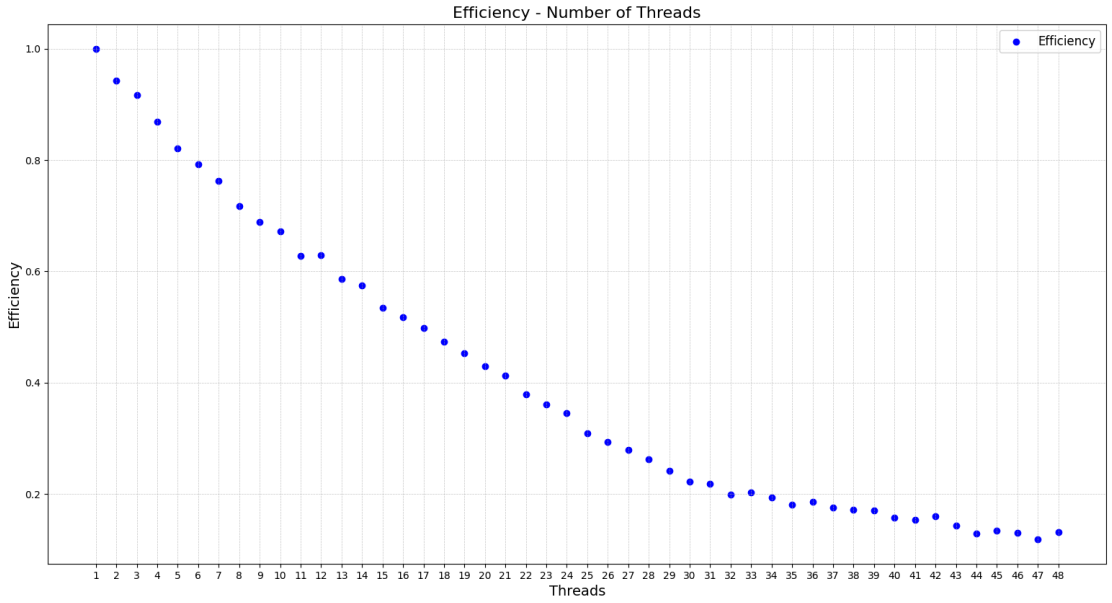

Fig. 3: Efficiency Graph (partition `dav`)

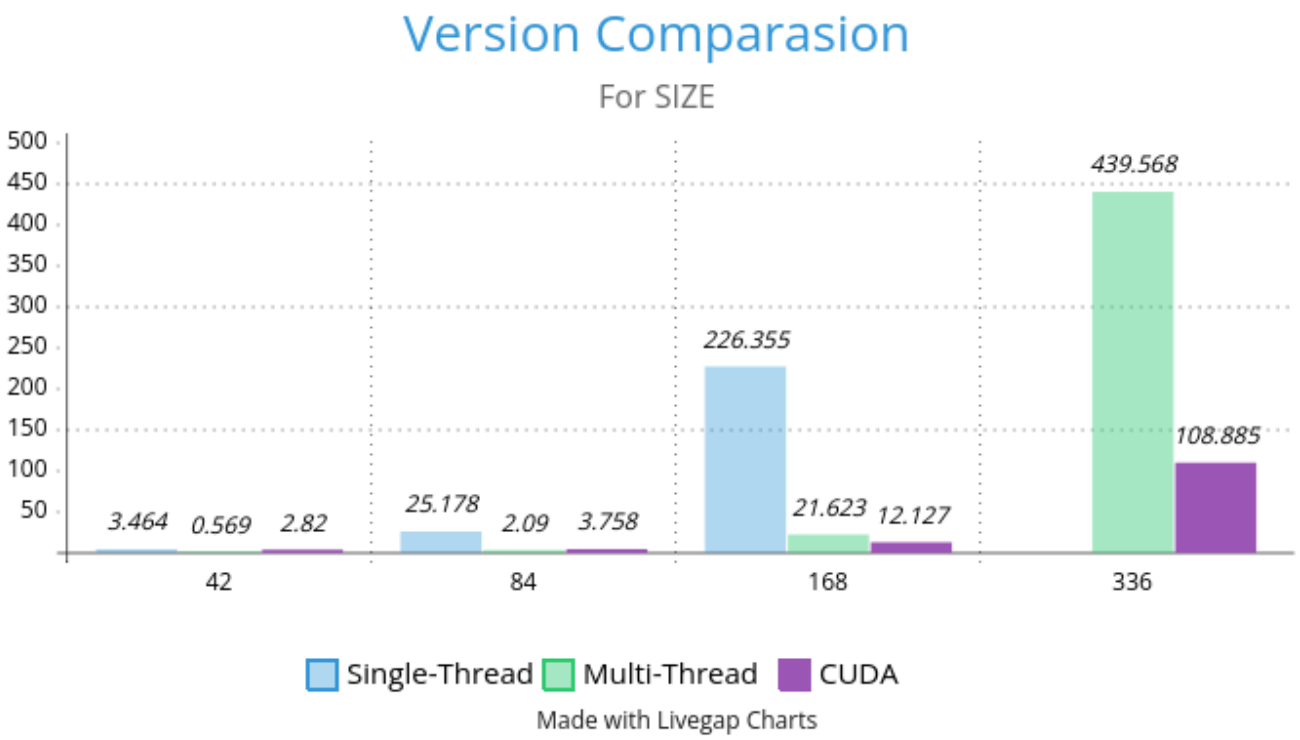

Fig. 4: Values in a table:

Single-Thread	3.464s	25.178s	226.355s	1200> s
Multi-Thread	0.569s	2.09s	21.623s	439.568s
CUDA	2.82s	3.758s	12.127s	108.885s

Times per version per size

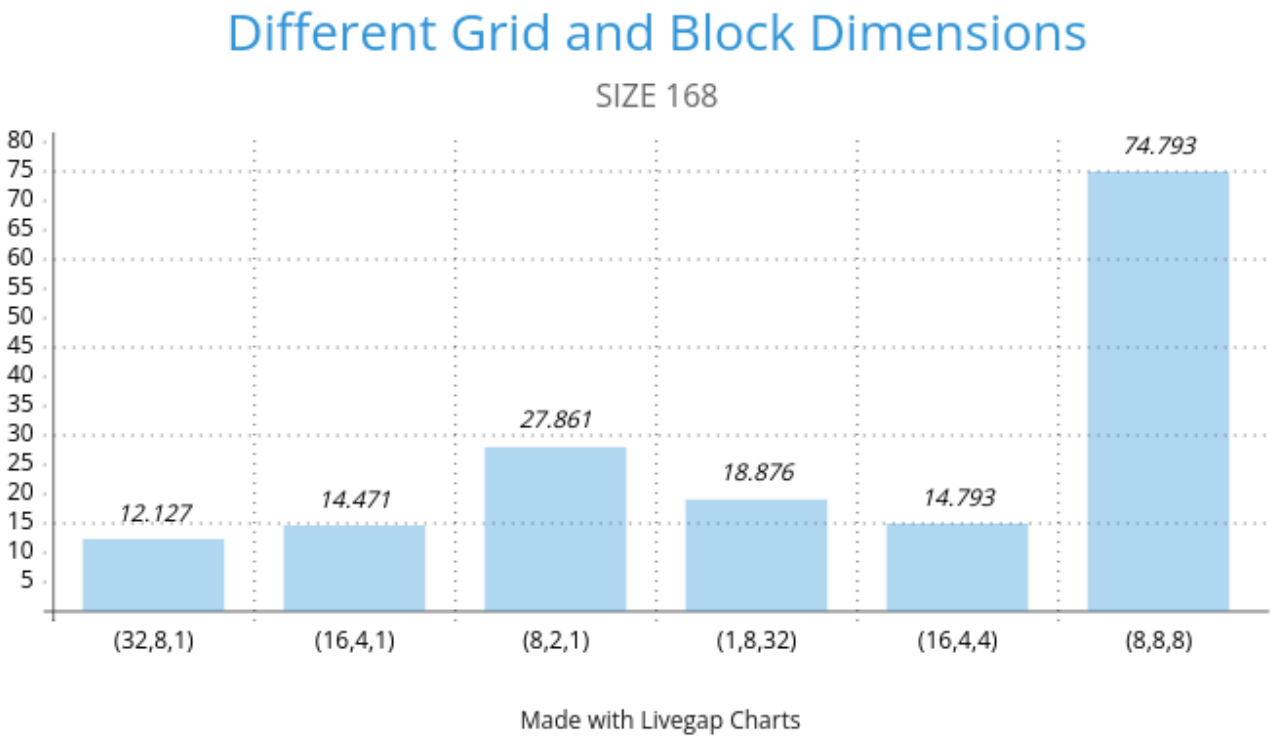

Fig. 5: Block Dimensions Tested