Projet Techniques d'Optimisation de la Parallélisation (TOP)

Seismic core - 3D stencil optimization

université paris Saclay
M1 CHPS

Rédigé par
Ahmed Taleb Bechir
Abdelghani Ameziane

Encadrant :
Gabriel Dos Santos
Hugo Taboada

1 Introduction

The project focuses on optimizing a seismic core algorithm that performs a 3D stencil computation. The algorithm involves a 16th-order, 49-point stencil operation, which is crucial for seismic data processing in various scientific and engineering applications.
The primary objective of the optimization effort is to enhance the computational efficiency of the algorithm while maintaining its accuracy and reliability. The optimization process aims to reduce the overall computational time required for the stencil computation, thereby enabling faster seismic data analysis and processing.
Key challenges in optimizing the algorithm include :
Memory Access Patterns : Efficient management of memory access patterns is essential to minimize data transfer overhead and maximize cache utilization during stencil computation. Parallelization : Leveraging parallel processing techniques such as multithreading or vectorization to distribute the computational workload across multiple processing units can significantly accelerate the stencil computation. Algorithmic Optimization : Exploring alternative algorithmic approaches or optimization strategies to minimize redundant computations and improve arithmetic intensity can lead to further performance gains. By addressing these challenges and implementing effective optimization techniques, the project aims to achieve substantial improvements in the performance and scalability of the seismic core algorithm, thereby facilitating faster and more efficient seismic data analysis and interpretation.

2 Environnement

Tous les tests ont été exécutés dans cet environnement :
— OS : Arch Linux x86 64 Kernel 6.5.7-arch1-1

— CPU : Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz

— RAM : 16GB

— Compilateur : gcc version 11.4.0 Ubuntu clang version 14.0.0-1
— profilers : perf strace gprof flamegraphs maqao

3 Optimization Steps

3.1 Initial Profiling and Analysis

Initially, the program encountered compilation errors. The CMake configuration needed correction to properly detect and compile MPI. Additionally, we identified issues with error management in the mesh initialization module. After addressing these issues, we achieved a functioning code that compiles successfully.
Our first step was to run the program with a 100x100x100 configuration file for 10 iterations to obtain an initial speed reading.
Subsequently, we employed the 'perf' tool to profile the code, recording metrics such as cycles, instructions, cache-references, and cache-misses. Additionally, we utilized 'gprof' to generate a call graph of the program.
Our initial optimization involved compiler settings. For the initial profiling, we did not utilize compiler optimization flags. However, subsequent runs were conducted with 'O3' and 'Ofast' optimization flags enabled.

```
Performance counter stats for './top-stencil ../config.txt':

13407600584      cycles
10991068847      instructions              #    0,82  insn per cycle
1957075053      cache-references
726958179      cache-misses              #   37,145 % of all cache refs
27,965110103 seconds time elapsed
4,563284000 seconds user
0,060095000 seconds sys

Result_no_flag initial AVERAGE = 4028.01
Result flag 03 AVERAGE=3614.29
Result flag ofast  Average = 2447.98
```

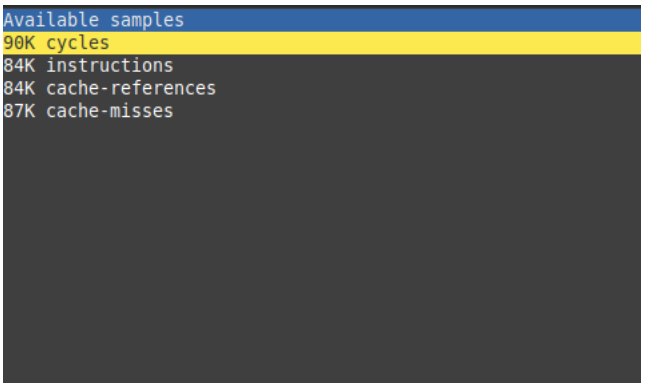
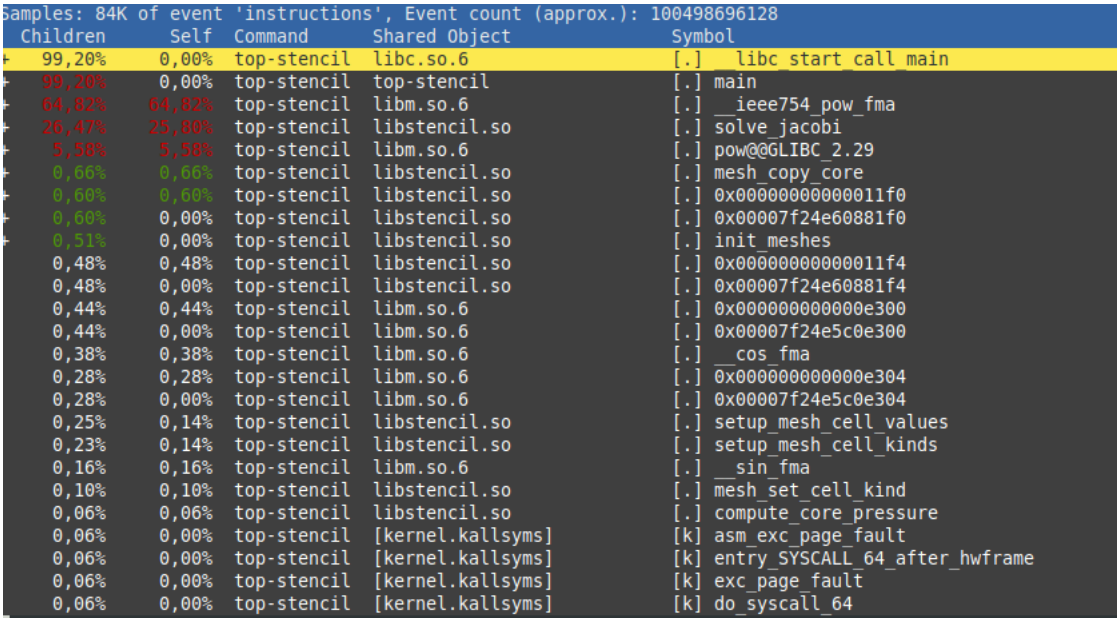


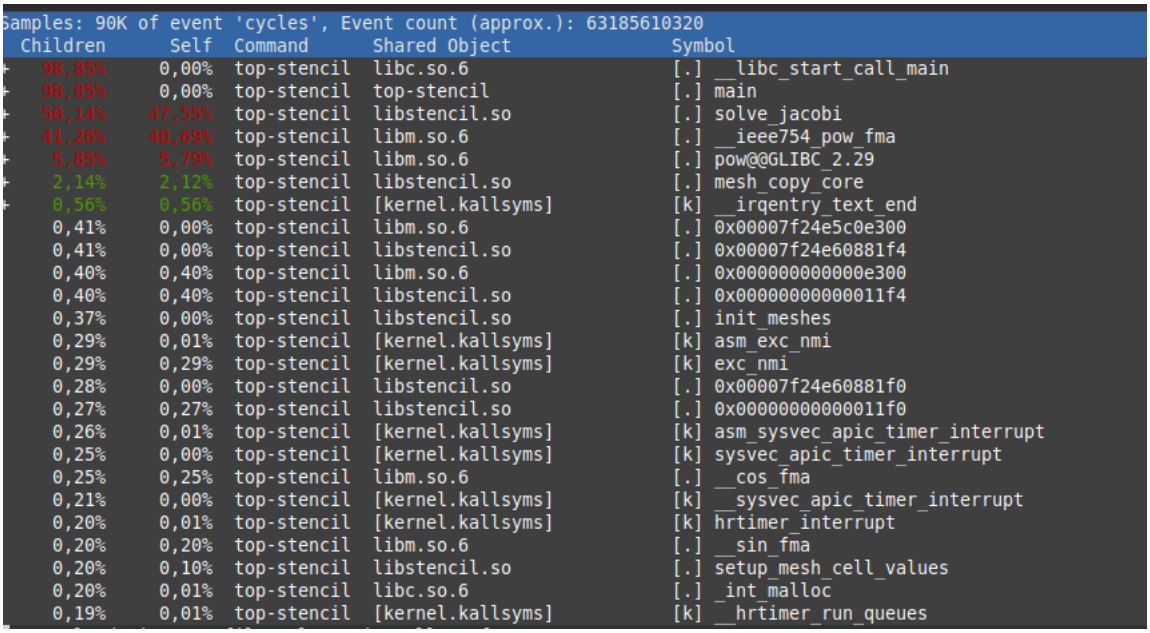
FIGURE 1 – perf screenshots

Reference is 60.32% faster than result.



```
Samples: 84K of event 'instructions', Event count (approx.): 100498696128
  Children      Self  Command      Shared Object               Symbol
+   99,20%     0,00%  top-stencil  libc.so.6                   [.] __libc_start_call_main
+   99,20%     0,00%  top-stencil  top-stencil                 [.] main
+   64,82%    64,82%  top-stencil  libm.so.6                   [.] __ieee754_pow_fma
+   26,47%    25,80%  top-stencil  libstencil.so               [.] solve_jacobi
+    5,58%     5,58%  top-stencil  libm.so.6                   [.] pow@@GLIBC_2.29
+    0,66%     0,66%  top-stencil  libstencil.so               [.] mesh_copy_core
+    0,60%     0,60%  top-stencil  libstencil.so               [.] 0x00000000000011f0
+    0,60%     0,00%  top-stencil  libstencil.so               [.] 0x00007f24e60881f0
+    0,51%     0,00%  top-stencil  libstencil.so               [.] init_meshes
     0,48%     0,48%  top-stencil  libstencil.so               [.] 0x00000000000011f4
     0,48%     0,00%  top-stencil  libstencil.so               [.] 0x00007f24e60881f4
     0,44%     0,44%  top-stencil  libm.so.6                   [.] 0x000000000000e300
     0,44%     0,00%  top-stencil  libm.so.6                   [.] 0x00007f24e5c0e300
     0,38%     0,38%  top-stencil  libm.so.6                   [.] __cos_fma
     0,28%     0,28%  top-stencil  libm.so.6                   [.] 0x000000000000e304
     0,28%     0,00%  top-stencil  libm.so.6                   [.] 0x00007f24e5c0e304
     0,25%     0,14%  top-stencil  libstencil.so               [.] setup_mesh_cell_values
     0,23%     0,14%  top-stencil  libstencil.so               [.] setup_mesh_cell_kinds
     0,16%     0,16%  top-stencil  libm.so.6                   [.] __sin_fma
     0,10%     0,10%  top-stencil  libstencil.so               [.] mesh_set_cell_kind
     0,06%     0,06%  top-stencil  libstencil.so               [.] compute_core_pressure
     0,06%     0,00%  top-stencil  [kernel.kallsyms]          [k] asm_exc_page_fault
     0,06%     0,00%  top-stencil  [kernel.kallsyms]          [k] entry_SYSCALL_64_after_hwframe
     0,06%     0,00%  top-stencil  [kernel.kallsyms]          [k] exc_page_fault
     0,06%     0,00%  top-stencil  [kernel.kallsyms]          [k] do_syscall_64
```

FIGURE 2 – perf screenshots



```
Samples: 90K of event 'cycles', Event count (approx.): 63185610320
  Children      Self  Command      Shared Object               Symbol
+   98,85%     0,00%  top-stencil  libc.so.6                   [.] __libc_start_call_main
+   98,85%     0,00%  top-stencil  top-stencil                 [.] main
+   50,14%    47,55%  top-stencil  libstencil.so               [.] solve_jacobi
+   41,26%    40,69%  top-stencil  libm.so.6                   [.] __ieee754_pow_fma
+    5,85%     5,79%  top-stencil  libm.so.6                   [.] pow@@GLIBC_2.29
+    2,14%     2,12%  top-stencil  libstencil.so               [.] mesh_copy_core
+    0,56%     0,56%  top-stencil  [kernel.kallsyms]          [k] __irqentry_text_end
     0,41%     0,00%  top-stencil  libm.so.6                   [.] 0x00007f24e5c0e300
     0,41%     0,00%  top-stencil  libstencil.so               [.] 0x00007f24e60881f4
     0,40%     0,40%  top-stencil  libm.so.6                   [.] 0x000000000000e300
     0,40%     0,40%  top-stencil  libstencil.so               [.] 0x00000000000011f4
     0,37%     0,00%  top-stencil  libstencil.so               [.] init_meshes
     0,29%     0,01%  top-stencil  [kernel.kallsyms]          [k] asm_exc_nmi
     0,29%     0,29%  top-stencil  [kernel.kallsyms]          [k] exc_nmi
     0,28%     0,00%  top-stencil  libstencil.so               [.] 0x00007f24e60881f0
     0,27%     0,27%  top-stencil  libstencil.so               [.] 0x00000000000011f0
     0,26%     0,01%  top-stencil  [kernel.kallsyms]          [k] asm_sysvec_apic_timer_interrupt
     0,25%     0,00%  top-stencil  [kernel.kallsyms]          [k] sysvec_apic_timer_interrupt
     0,25%     0,25%  top-stencil  libm.so.6                   [.] __cos_fma
     0,21%     0,00%  top-stencil  [kernel.kallsyms]          [k] __sysvec_apic_timer_interrupt
     0,20%     0,01%  top-stencil  [kernel.kallsyms]          [k] hrtimer_interrupt
     0,20%     0,20%  top-stencil  libm.so.6                   [.] __sin_fma
     0,20%     0,10%  top-stencil  libstencil.so               [.] setup_mesh_cell_values
     0,20%     0,01%  top-stencil  libc.so.6                   [.] _int_malloc
     0,19%     0,01%  top-stencil  [kernel.kallsyms]          [k] __hrtimer_run_queues
```

FIGURE 3 – perf screenshots

```
Samples: 87K of event 'cache-misses', Event count (approx.): 10143545
  Children      Self  Command      Shared Object                Symbol
+   99,41%     0,00%  top-stencil  libc.so.6                    [.] __
+   99,41%     0,00%  top-stencil  top-stencil                  [.] ma
+   47,98%    42,10%  top-stencil  libstencil.so                [.] so
+   44,85%    44,65%  top-stencil  libm.so.6                    [.] __
+    5,66%     5,64%  top-stencil  libstencil.so                [.] me
+    5,12%     5,09%  top-stencil  libm.so.6                    [.] po
     0,40%     0,00%  top-stencil  [kernel.kallsyms]            [k] as
     0,39%     0,00%  top-stencil  [kernel.kallsyms]            [k] sy
     0,35%     0,00%  top-stencil  libstencil.so                [.] in
     0,35%     0,00%  top-stencil  [kernel.kallsyms]            [k] __
     0,34%     0,00%  top-stencil  [kernel.kallsyms]            [k] hr
     0,34%     0,34%  top-stencil  libstencil.so                [.] 0x
     0,34%     0,00%  top-stencil  libstencil.so                [.] 0x
     0,33%     0,01%  top-stencil  [kernel.kallsyms]            [k] __
     0,30%     0,00%  top-stencil  libstencil.so                [.] 0x
     0,30%     0,30%  top-stencil  libstencil.so                [.] 0x
     0,29%     0,00%  top-stencil  [kernel.kallsyms]            [k] ti
     0,27%     0,00%  top-stencil  libm.so.6                    [.] 0x
     0,26%     0,26%  top-stencil  libm.so.6                    [.] 0x
     0,20%     0,20%  top-stencil  libm.so.6                    [.] 0x
     0,20%     0,00%  top-stencil  libm.so.6                    [.] 0x
     0,19%     0,00%  top-stencil  [kernel.kallsyms]            [k] ti
     0,19%     0,01%  top-stencil  [kernel.kallsyms]            [k] up
     0,17%     0,11%  top-stencil  libstencil.so                [.] se
     0,17%     0,00%  top-stencil  [kernel.kallsyms]            [k] as
```

FIGURE 4 – perf screenshots

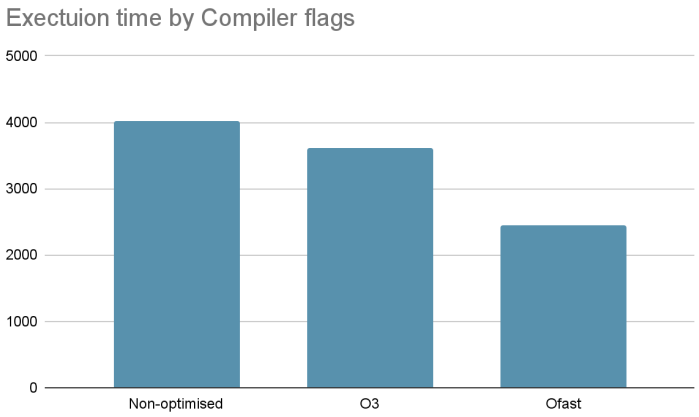


FIGURE 5 – compiler flags comparison

FIGURE 6 – flamegraph visualisation

3.1.1 Optimization Attempt 2 : using Strace to remove obsffuction of malicious code

so for our second attempt at optimizing the code, we corrected some bugs we didn't see in the first phase, such a bug that didn't allow other ranks to access the all reduce mp function in the main. This allowed the field in our results to be corrected.
We stuck with the Ofast flag for the rest of the project. We ran strace on the program and found a small little system call that called sleep(), which obviously was contributing massively to our performance drag.
Result R2 ofast = 362.9053

```
Performance counter stats for './top-stencil ../config.txt':
    12088115629      cycles
    10987954787      instructions              #    0,91  insn per cycle
     1949147064      cache-references
       681035780       cache-misses             #   34,940 % of all cache refs
       4,391701852 seconds time elapsed
       4,028489000 seconds user
       0,048149000 seconds sys
```

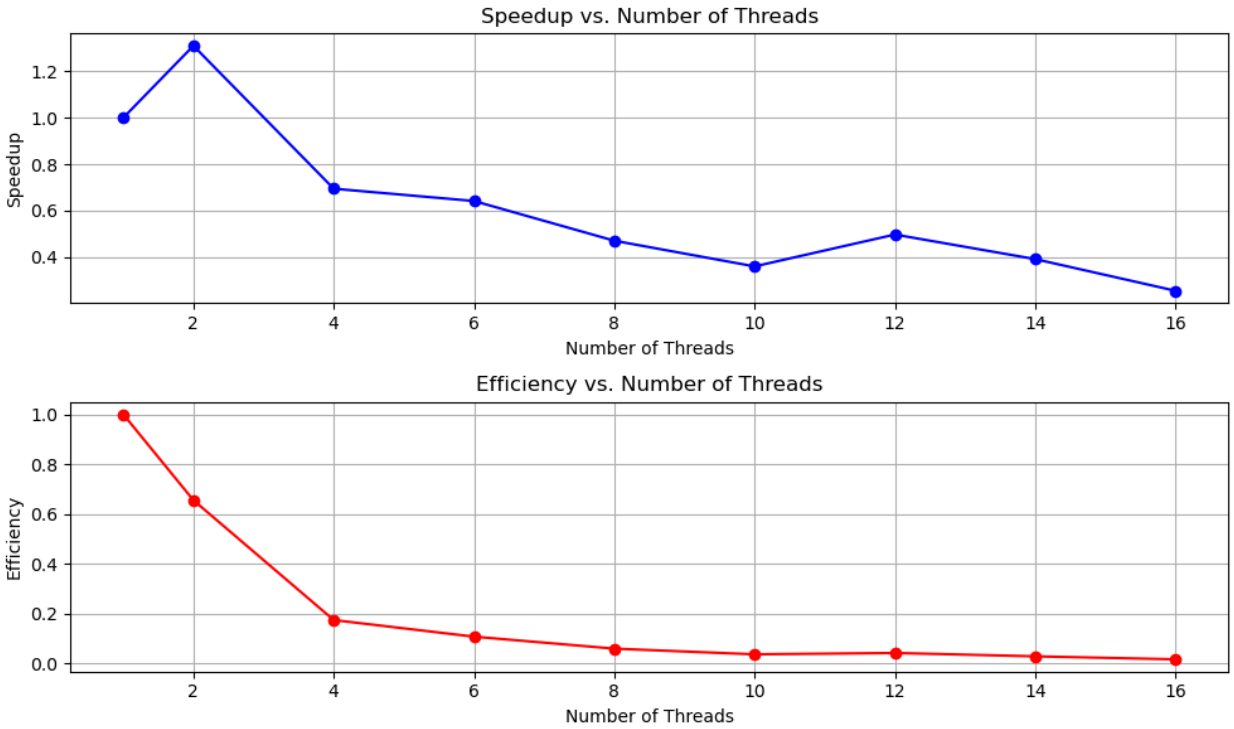


FIGURE 7 – speed-up/by number of threadq

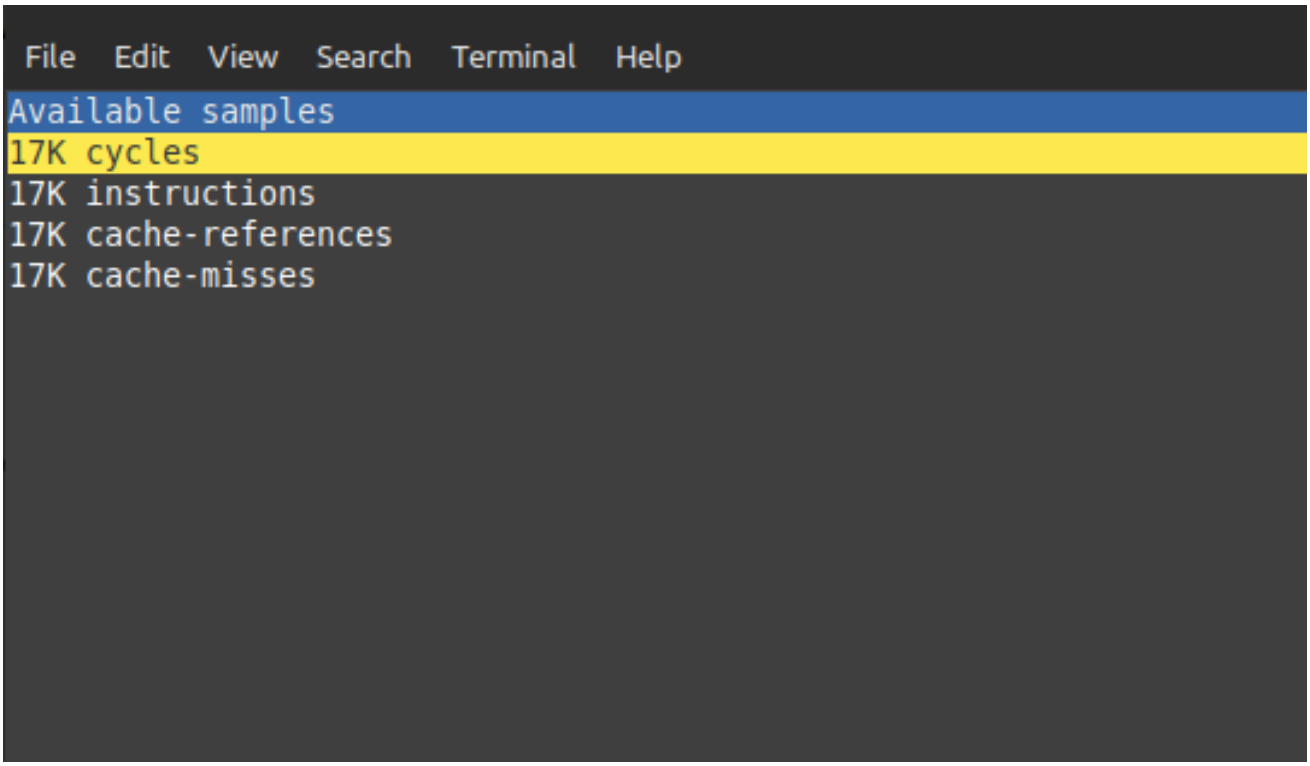


FIGURE 8 – perf screenshots

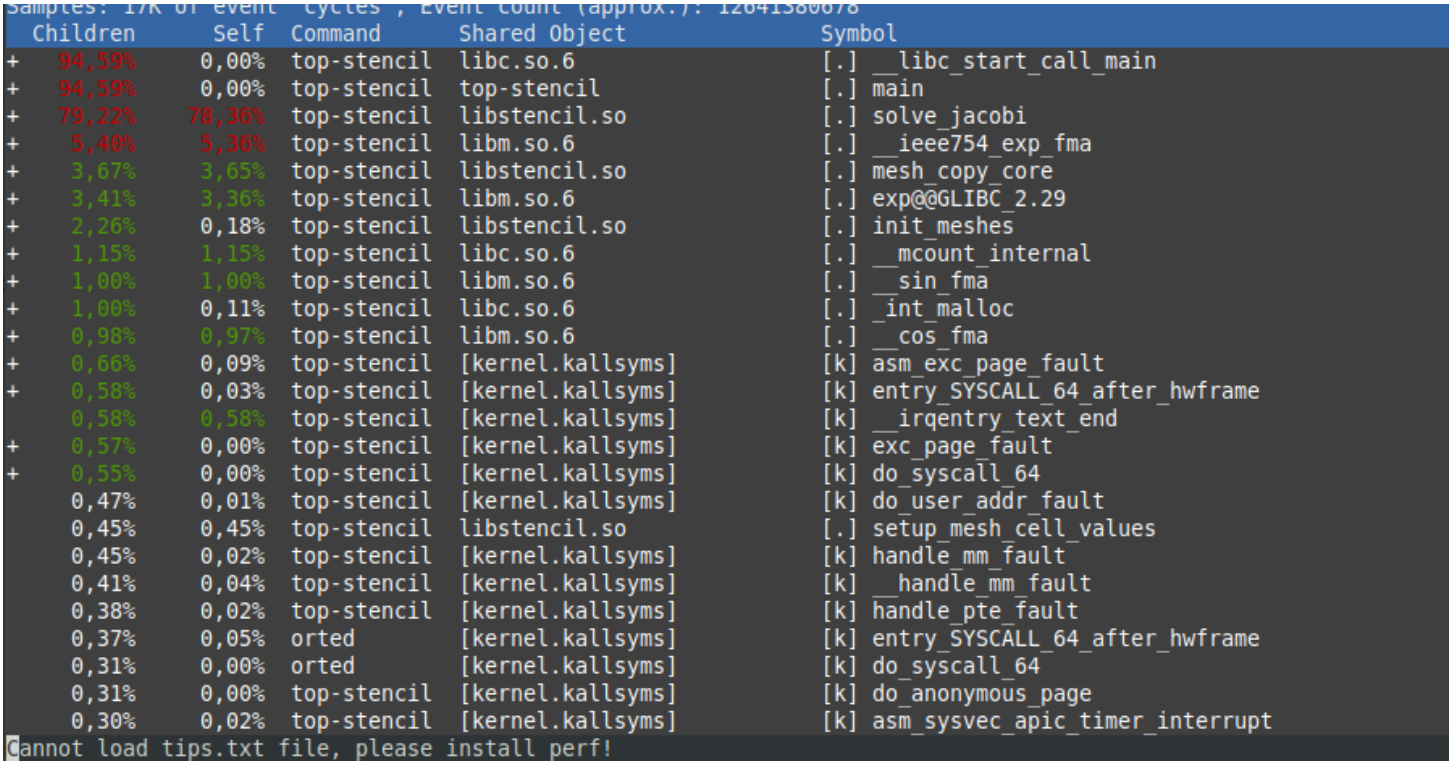


FIGURE 9 – perf screenshots

```
Samples: 17K of event 'cache-misses', Event count (approx.): 694745482
  Children      Self  Command      Shared Object               Symbol
+   99,22%     0,00%  top-stencil  libc.so.6                   [.] __libc_start_call_main
+   99,22%     0,00%  top-stencil  top-stencil                 [.] main
+   83,45%    83,20%  top-stencil  libstencil.so               [.] solve_jacobi
+    7,33%     7,31%  top-stencil  libm.so.6                   [.] __ieee754_exp_fma
+    3,81%     3,78%  top-stencil  libm.so.6                   [.] exp@@GLIBC_2.29
+    3,71%     3,69%  top-stencil  libstencil.so               [.] mesh_copy_core
     0,39%     0,04%  top-stencil  libstencil.so               [.] init_meshes
     0,31%     0,00%  top-stencil  [kernel.kallsyms]           [k] asm_sysvec_apic_timer_interrupt
     0,31%     0,00%  top-stencil  [kernel.kallsyms]           [k] sysvec_apic_timer_interrupt
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] asm_exc_page_fault
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] exc_page_fault
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] do_user_addr_fault
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] handle_mm_fault
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] __handle_mm_fault
     0,23%     0,00%  top-stencil  [kernel.kallsyms]           [k] handle_pte_fault
     0,21%     0,01%  top-stencil  libc.so.6                   [.] _int_malloc
     0,21%     0,00%  top-stencil  [kernel.kallsyms]           [k] do_anonymous_page
     0,21%     0,21%  top-stencil  libc.so.6                   [.] __mcount_internal
     0,19%     0,19%  top-stencil  libstencil.so               [.] setup_mesh_cell_values
     0,17%     0,01%  top-stencil  [kernel.kallsyms]           [k] __sysvec_apic_timer_interrupt
     0,17%     0,00%  top-stencil  [kernel.kallsyms]           [k] hrtimer_interrupt
     0,16%     0,00%  top-stencil  [kernel.kallsyms]           [k] __hrtimer_run_queues
     0,16%     0,16%  top-stencil  libm.so.6                   [.] 0x000000000000e380
     0,16%     0,00%  top-stencil  libm.so.6                   [.] 0x00007f7ae8f37380
     0,15%     0,00%  top-stencil  [kernel.kallsyms]           [k] entry_SYSCALL_64_after_hwframe
```

FIGURE 10 – perf screenshots

```
ercent      B->cells[i][j - o][k].value / pow(17.0, (f64)o);
 0,53         mov      -0x48(%r8,%r12,8),%r9
 0,62         mov      %r13,%r8
            for (usz o = 1; o <= STENCIL_ORDER; ++o) {
              add      $0x1,%r13
 0,12         shl      $0x4,%r8
            B->cells[i - o][j][k].value / pow(17.0, (f64)o);
 0,28         mulsd    %xmm2,%xmm3
            C->cells[i][j][k].value += A->cells[i - o][j][k].value *
 0,66         addsd    %xmm3,%xmm7
 0,78         movsd    %xmm7,(%r14)
            C->cells[i][j][k].value += A->cells[i][j + o][k].value *
 0,55         movsd    (%r11,%rbx,1),%xmm4
11,73         mulsd    (%rax,%rbx,1),%xmm4
10,03         mov      %r12,%r11
 0,38         shl      $0x4,%r11
            B->cells[i][j + o][k].value / pow(17.0, (f64)o);
 0,13         mulsd    %xmm2,%xmm4
            C->cells[i][j][k].value += A->cells[i][j + o][k].value *
 1,60         addsd    %xmm7,%xmm4
 2,28         movsd    %xmm4,(%r14)
            C->cells[i][j][k].value += A->cells[i][j - o][k].value *
 0,68         movsd    (%r10,%rbx,1),%xmm1
 3,57         mulsd    (%r9,%rbx,1),%xmm1
```

FIGURE 11 – perf screenshots

```
plotting cache misses compared to the first version .
cache misses for opt1:17K
cache misses for opt2:87K
plotting cycles compared to the first version :
execution cycles for opt1 : 17K
execution cycles for opt2 : 90K
```

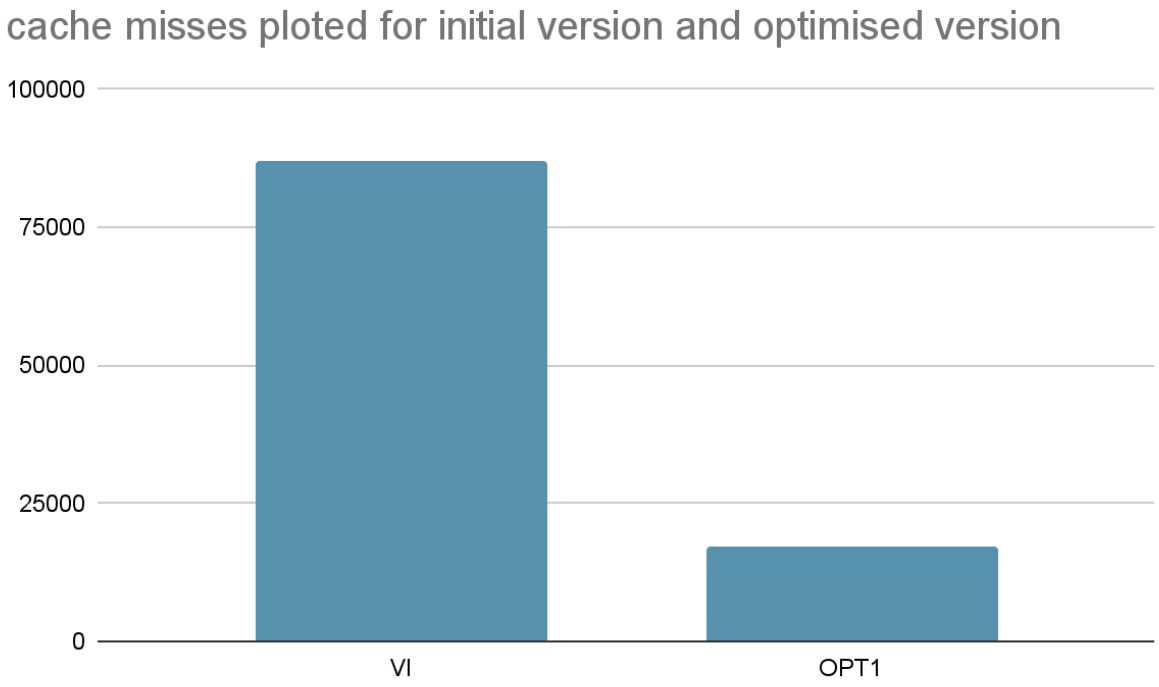


FIGURE 12 – cache misses ploted

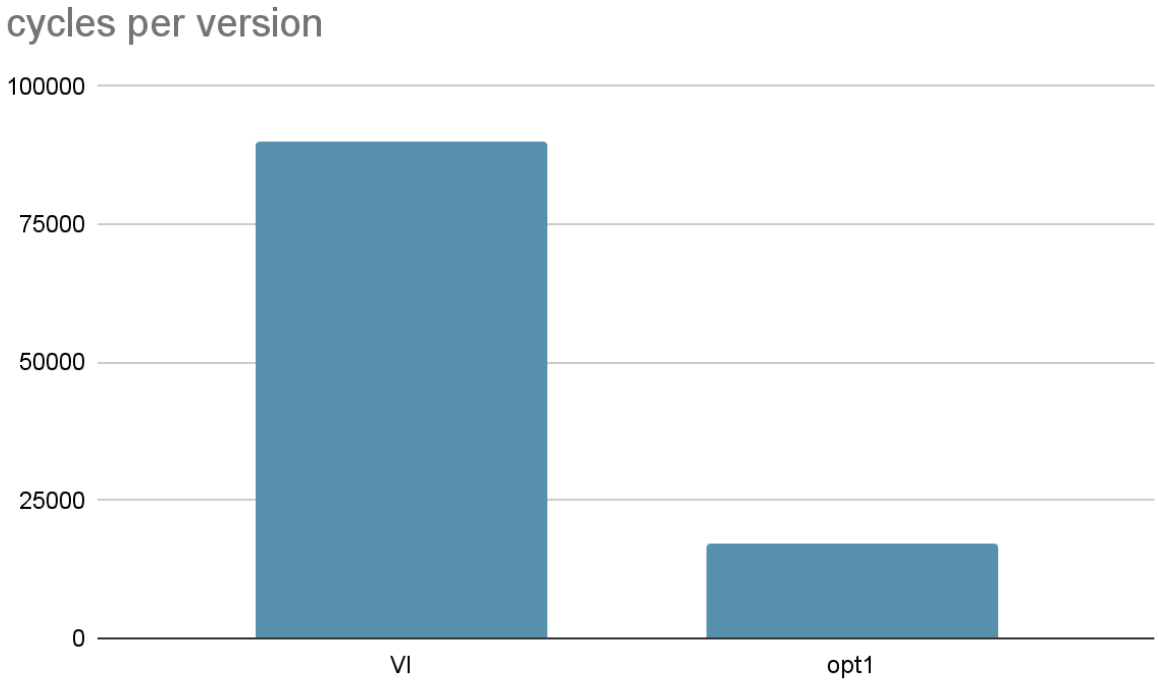


FIGURE 13 – cycles by optim

Result is 140.31% faster than Reference

3.1.2 Optimization Attempt 3 : Precomputing Pows

in our third optimization we wanted to reduce the cost of pow operations , so we precomputed the pows in a loop table (array) and just accessed hem along the mesh access
we recorded a small gain in performance.
R3 Ofast : Average = 336.3882

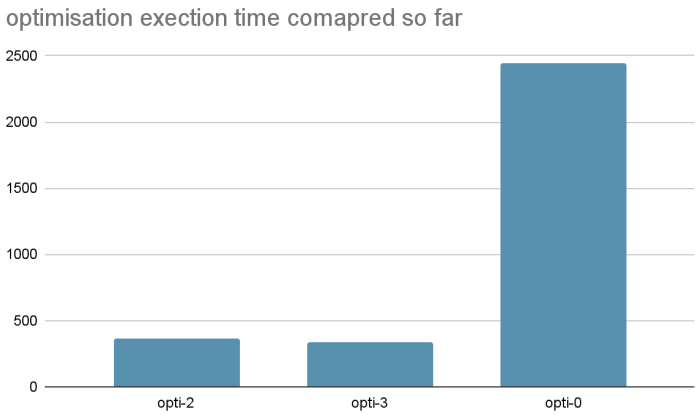


FIGURE 14 – Optimization comparison

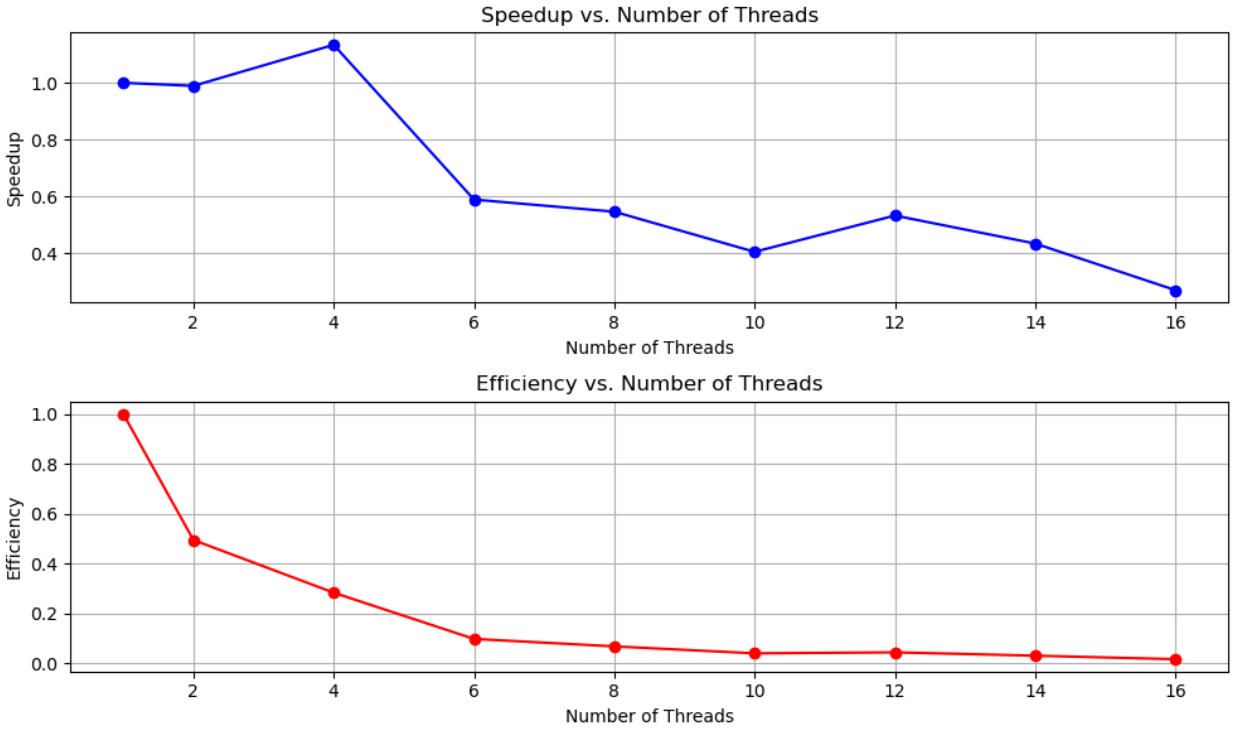


FIGURE 15 – scaling plot

Result is 199.43% faster than reference

3.1.3 Optimization Attempt 4 : Specific Pow Implementation

We took on from earlier, and we decided to optimize the pow function to see if there was a performance gain to accompany the pre-computation f the values. We did also gain a small chunk of performance from the specific pow implementation.
R4 Ofast Average = 298.5749

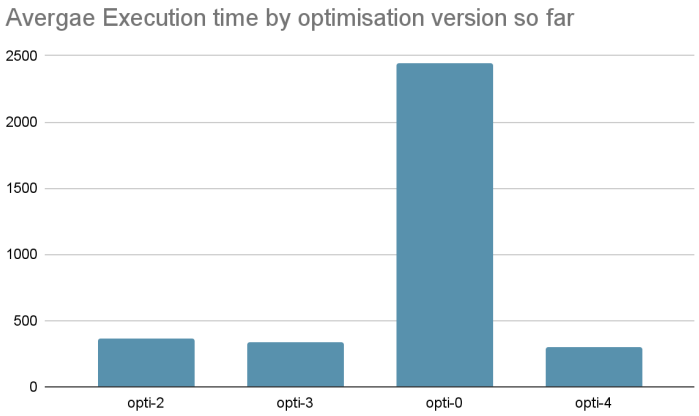


FIGURE 16 – scaling plot

Result is 253.37%faster than reference

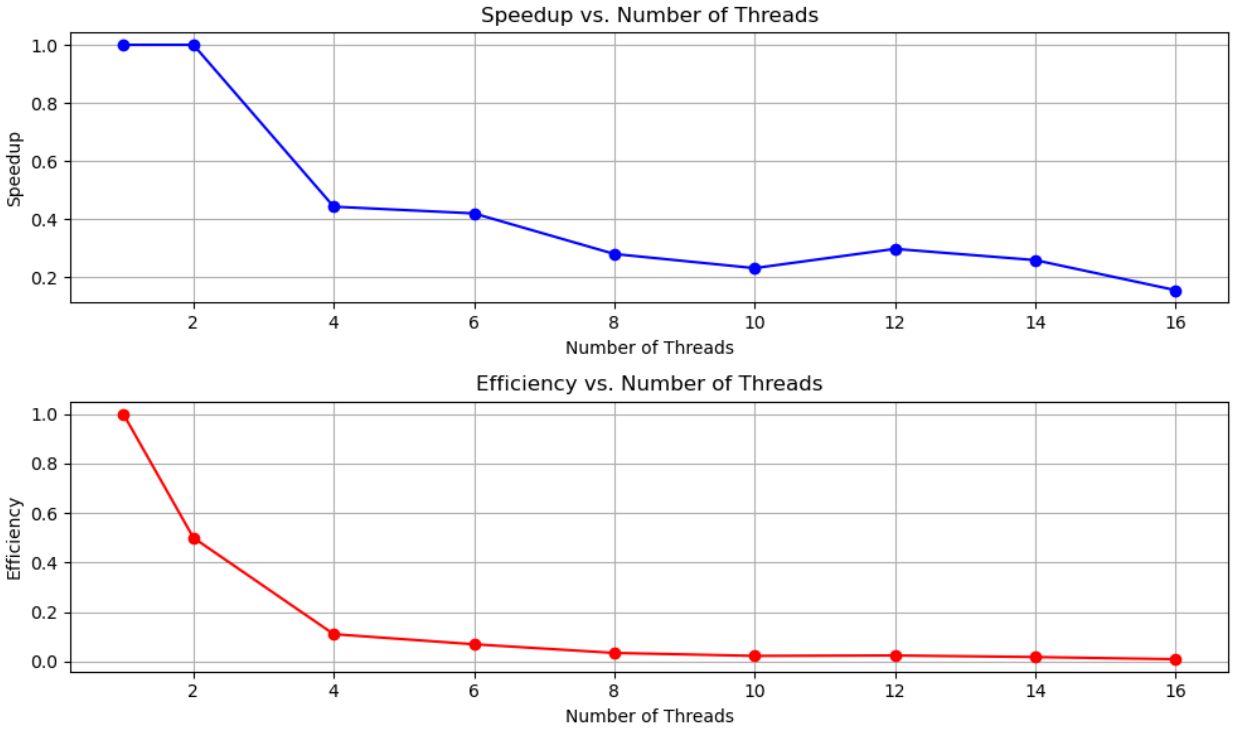


FIGURE 17 – scaling plot

3.1.4 Optimization Attempt 5 : implementing Cache blocking

Cache blocking is a fundamental optimization technique crucial for enhancing memory access patterns and reducing cache misses. By breaking down data into smaller blocks that fit efficiently into cache levels, cache blocking maximizes cache utilization and minimizes cache miss rates.
In our optimization endeavors, we integrated cache blocking techniques into our 5th attempt. Specifically, we adopted blocks of size 64 to mitigate cache misses and improve memory locality. This strategic optimization led to a discernible enhancement in performance, streamlining memory access patterns and reducing cache misses.
We varied the cache size from and measured the execution time. We found with size 16 we had the best average execution time and reduction in cache misses , the blocks were similar in all direction (x,y,z) , although we tried to experiment with a different access pattern such as 8 blocks in the X direction and 4 in both Y and Z directions and also an 8/1/1 split but it was much less effective for our problem size, it might be interesting in larger problems .
R5 Ofast Average with Cache size 64 = 218.8748 .
with cache size 8 we were able to obtain Average = 173.7580.
with cache size 4 the Average = 189.44
with cache size 16 the average is 169.1500
with cache size 32 the average is 200.8483
with cache size split of 8/4/4 the average is 193.3652
with cache size split of 8/1/1 the Average is 295.3421
We ran perf on this version and recorded these measures :

```
Performance counter stats for './top-stencil ../config.txt':

     5662365742      cycles
     5031504709      instructions              #    0,89  insn per cycle
       403648349       cache-references
       175646578       cache-misses              #   43,515 % of all cache refs
       2,252407360 seconds time elapsed
       1,846213000 seconds user
       0,068229000 seconds sys
```

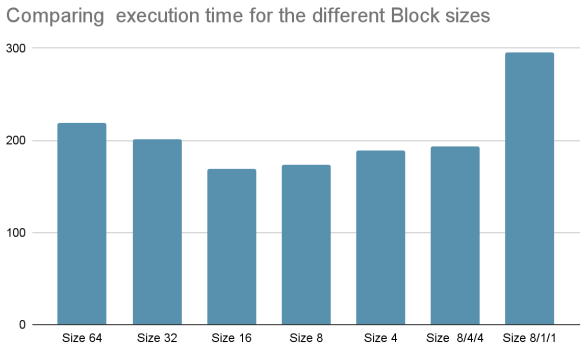


FIGURE 18 – block size comparison for cache-blocking

3.1.5 Optimization Attempt 6 : Rearranging Mesh Data structure

in this optimization attempt, we restructured the mesh data from a Structure of Arrays (SoA) to an Array of Structures (AoS). This change in data structure organization aimed to enhance memory locality and improve cache utilization.
After implementing the new AoS data structure, we observed a notable boost in performance. Specifically, when running with the 'Ofast' compiler optimization flag (referred to as 'R6 ofast'), we achieved an average execution time of 63.9168.

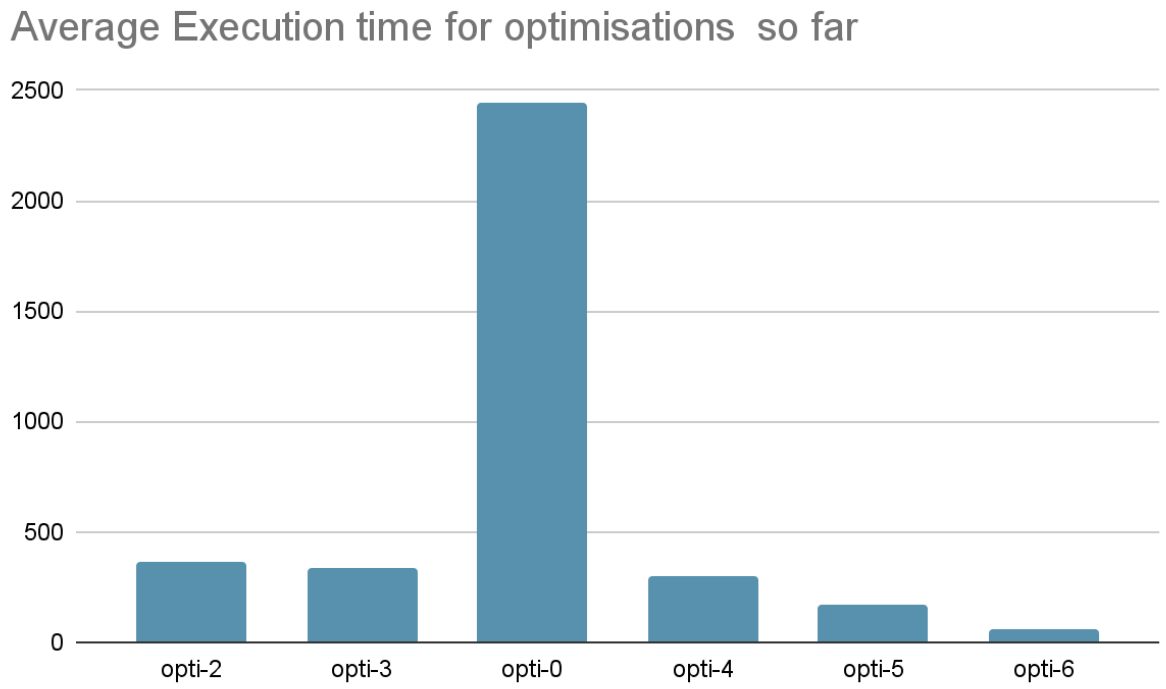


FIGURE 19 – execution time plots

3.1.6 Optimization Attempt 7 : using openmp

We used openmp in the solve jacobi to distribute the main loop. using the openmp collapse clause, which Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause

```
execution time per number of thread.
 2 threads: 54.9468
 4 threads: 39.7711
 6 threads: 45.7734
 8 threads: 43.7071
10 threads:45.0123
12 threads: 45.0981
14 threads: 43.747
16 threads: 44.783
```

Result is 2820.83% faster than reference

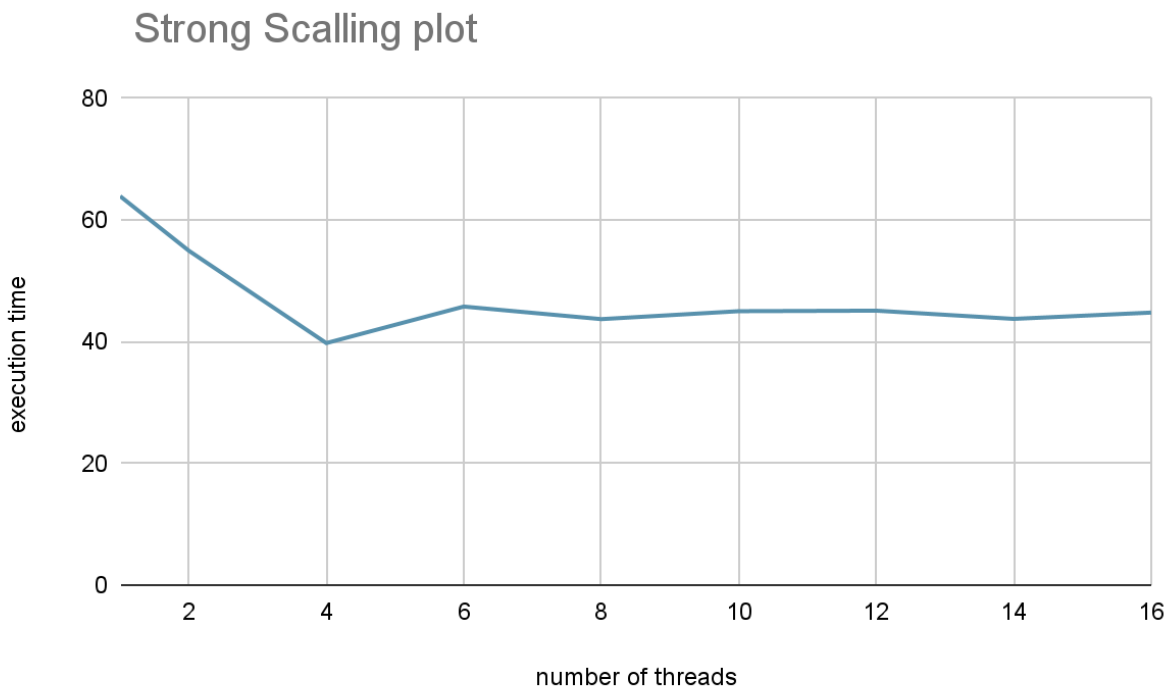


FIGURE 20 – scaling plot

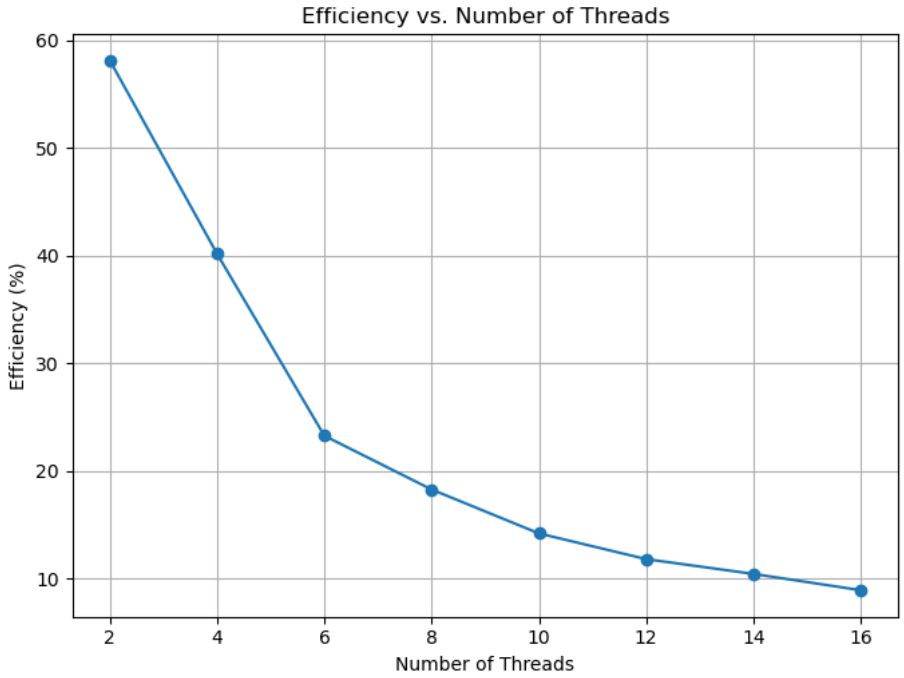


FIGURE 21 – efficiency/speed-up plot

4 Conclusion

In this report, we embarked on a journey to optimize the performance of our computational program through systematic strategies and targeted optimizations. Our efforts began by addressing compilation errors and optimizing compiler settings, laying the groundwork for subsequent optimization attempts.
Profiling tools such as 'perf' and 'gprof' provided invaluable insights into the program's performance characteristics, guiding our optimization endeavors. Through careful analysis of performance metrics, we identified key areas for improvement and devised targeted optimization strategies.
One notable optimization attempt involved restructuring the mesh data from a Structure of Arrays (SoA) to an Array of Structures (AoS). This data structure rearrangement aimed to enhance memory access patterns and cache utilization, resulting in a significant boost in performance. Leveraging compiler optimization flags, further contributed to the efficiency gains, culminating in a speedup of 2820.83
Our optimization journey underscores the importance of systematic optimization strategies in improving program performance. By addressing compilation errors, optimizing compiler settings, and leveraging profiling tools, we successfully enhanced the efficiency and speed of the program without radically changing the initial code. By changing the strategies we use to access and manipulate data we were able to see a huge jump in performance, and by going through the multiple techniques from removing redundancies and utilizing our cache line more effectively, we learned that the optimizations that net the biggest gain aren't always the most extravagant and complicated, but ares imply the ones that show a fundamental understanding of what were trying to achieve with our code whether its simulating natural phenomenons or visualizing observations, It's important to leverage the understanding of your environment and its parameters. Due to time constraints, we left a chunk of performance on the table, especially with vectorizing our code to leverage 128 and 256 bit registers. These optimizations demonstrate the effectiveness of considering various factors such as memory access patterns, cache efficiency, and parallels scaling in achieving performance improvements in computational tasks.

Références

[1] Zhao, Wenxuan, et al. "Stencil Matrixization." (2023).

[2] "Vectorization and Parallelization of Loops in C / C + + Code." (2017).

[3] Fang, J. et al., "Optimizing Complex Spatially-Variant Coefficient Stencils for Seismic Modeling on GPU," 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, VIC, Australia, 2015, pp. 641-648, doi : 10.1109/ICPADS.2015.86.

[4] Kamil, Shoaib, et al. "Implicit and explicit optimizations for stencil computations." In Proceedings of the 2006 workshop on Memory system performance and correctness (MSPC '06). Association for Computing Machinery, New York, NY, USA, 2006. 51–60. https ://doi.org/10.1145/1178597.1178605

[5] Cattaneo, Riccardo, et al. "On How to Accelerate Iterative Stencil Loops : A Scalable Streaming-Based Approach." ACM Trans. Archit. Code Optim. 12, 4, Article 53 (January 2016), 26 pages. https ://doi.org/10.1145/2842615

[6] Datta, Kaushik, et al. "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors." SIAM Review 51, no. 1 (2009) : 129–59. http ://www.jstor.org/stable/20454196.

[7] Yount, Chuck. (2015). Vector Folding : Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. 10.1109/HPCC-CSS-ICESS.2015.27.