Blocking strategies for weather and climate models

High-Performance Computing for Weather and Climate Report

Yin Hau Lam, Duan-Heng Chang, Hei Tung Wu, Kam Lam Yeung

September 11, 2023

Abstract

Different blocking strategies are applicable to stencil code designed for weather and climate models in languages like Fortran or C/C++. While employing the k-blocking strategy is usually feasible and causes minimal disturbance to existing Fortran code, considering ij-blocking presents an intriguing option, which is utilized in the GT4Py DSL x86 backend. In this study, we are implementing k-blocking and ij-blocking techniques on a stencil program tailored for diffusion and contrasting how variations in block size relate to changes in computation time when different blocking strategies are used.

ETH zürich

1 Introduction

In the realm of high-performance computing and parallel processing, blocking strategies encompass various distinct methodologies. The selection of particular strategies is contingent upon factors such as the computational characteristics, the data structures utilized, and the intended hardware architecture. The most commonly used blocking strategies are k-blocking and ij-blocking. While k-blocking divides three-dimensional arrays along their third dimension (k), ij-Blocking divides matrices into smaller submatrices along rows (i) and columns (j). Both blocking strategies can reduce the loading of data from RAM by keeping them in caches and thus decrease the latencies.

1.1 Data arrays in memory

The structure of the data array in memory affects the ways a computer utilises its cache. Considering a field array that is stored in memory which has a type of real (kind=4), which is equal to float32 or a floating-point value that occupies 4 bytes in memory. The arrangement of data within this array is sequential, illustrated in Figure 1. In the case of a two-dimensional array, the first element of the second row is contiguous with the last element of the first row, and this pattern continues, depicted in Figure 2.

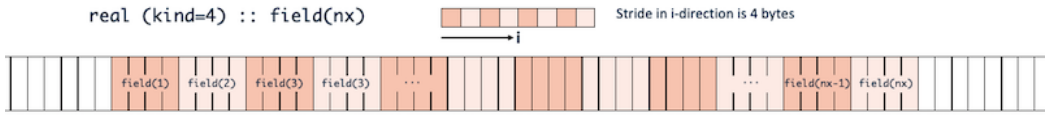

Figure 1. Memory alignment of a 1D array. (Fuhrer, 2023)

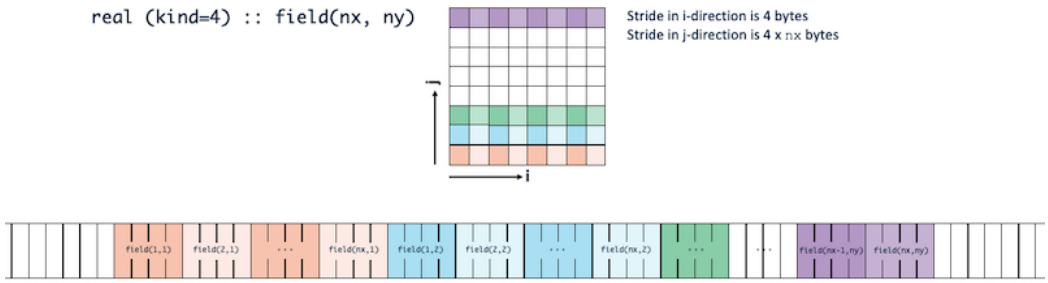

Figure 2. Memory alignment of a 2D array. (Fuhrer, 2023)

For a three-dimensional array utilized within the stencil code used in this study, denoted as field(i, j, k), the adjacency of field(1, 1, 1) and field(2, 1, 1) defines what is referred to as an i-stride. This i-stride signifies the transition from field(i, j, k) to field(i+1, j, k). Similarly, the separation between field(1, 1, 1) and field(1, 2, 1) is termed a j-stride, indicating the shift from field(i, j, k) to field(i, j+1, k). Lastly, the distinction between field(1, 1, 1) and field(1, 1, 2) represents a k-stride, signifying the progression from field(i, j, k) to field(i, j, k+1). The relationship between the dimension and the value is shown in Table 1.

Dimension	Value (for 128×128×64)
i-stride	4 Bytes
j-stride	512 Bytes (4 x nx)
k-stride	64 KBytes (4 x nx x ny)
full field	4 MBytes (4 x nx x ny x nz)

Table 1. Dimensions and memories used.

1.2 Cache Optimization

Latencies refer to the time taken to load a random element from a specific element of the memory hierarchy. Modern CPUs have a hierarchy of cache memories, as shown in Figure 3, aimed at hiding latencies associated with reading/writing operations to/from the primary memory. An L1, L2 and L3 cache have sizes of 32 KB per core, 256 KB per core and 30 MB per core respectively, while the RAM has a size of 64 GB. When all cores are engaged, there will only be 2.5 MB per core in the L3 cache. The constrained bandwidth leads to latency issues as data moves between cores, caches, and RAM. As indicated in Table 2, the most significant latencies arise when transferring data between the RAM and the CPU. Thus, if a substantial portion of the data can be retained within the cache, it becomes a practical approach for mitigating latencies, which is also the aim of blocking in this study.

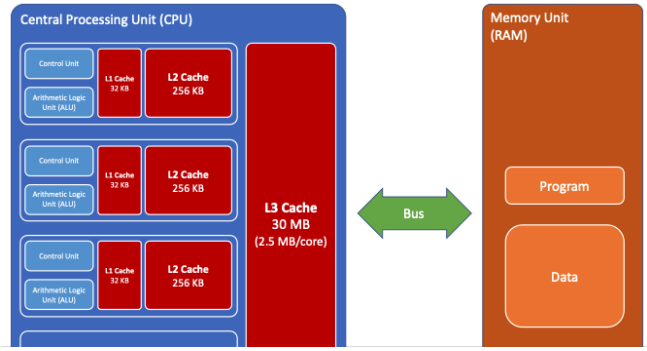

Figure 3. The cache hierarchy. (Fuhrer, 2023)

Link	Bandwidth	Latency
L1 ↔ core	read 155 GB/s + write 77.5 GB/s per core	≥ 4 cycles
L1 ↔ L2	155 GB/s per core	≥ 11 cycles
L2 ↔ L3	77.5 GB/s	≥ 36 cycles
L3 ↔ RAM	68.3 GB/s	≥ 100 cycles

Table 2. Bandwiths and latencies on Intel E5-2690 v3 Processor between the different elements of the memory hierarchy

2 Methodology

2.1 Stencil Program

The stencil program is developed in Fortran and is initially present in a file named *"stencil2d-orig.F90"* (Fuhrer, 2023). It applies diffusion and can be executed on all 12 cores of the Xeon E5-2690 v3 Haswell CPU utilized in this study. Numerous variables can be defined using command-line arguments, including *nx*, *ny* and *nz* to determine the computational domain's dimensions, as well as *num_iter* for specifying the iteration count. In our adapted stencil code for ij-blocking, the parameters *mx* and *my* can also be configured using command-line arguments to set the dimensions of an ij-block.

The original stencil program consists of a total of 1 main function and 8 subroutines. The utilization of these subroutines is outlined in the accompanying Table 3. Our modification of blocking mainly focuses on subroutine *"apply_diffusion"* and *"laplacian"*.

The stencil program generates the initial field and the resulting field after completing the iterations respectively. To confirm the application of diffusion, the data is imported into Python and visually verified. Figure 4 is an example of the visualization of diffusion outcomes of the original stencil program within a domain of dimensions (nx, ny, nz) = (128, 128, 64) after 1024 iterations, specifically at the

Subroutine	Utility
apply_diffusion	Integrate the 4th-order diffusion equation by a certain number of iterations.
laplacian	Compute Laplacian using 2nd-order centered differences.
update_halo	Update the halo-zone using an up/down and left/right strategy.
init	Initialize at program start.
setup	Setup everything before work.
read_cmd_line_arguments	Read and parse the command line arguments.
cleanup	Cleanup at the end of work.
finalize	Finalize at the end of the program.

Table 3. Utilization of subroutine in the stencil program.

middle level (k=32). In the following Sections, experiments were performed using nz = 4 with 128 iterations because of the computational costs.

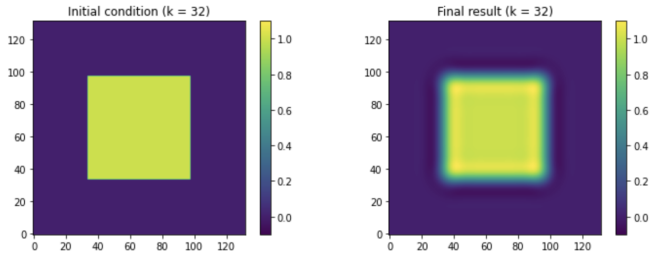

Figure 4. The diffusion outcomes of the original stencil program within a domain of dimensions (128, 128, 64) after 1024 iterations, specifically at k=0.

2.2 K-blocking

Instead of computing the entire domain step by step, k-blocking subdivides the domain into individual blocks along the k-dimension and processes them one by one. In the context of k-blocking, there is no overlap between the boundaries of these blocks, which are slices of x-y arrays. In this case, the necessary data remains within the CPU caches, eliminating the need for data transfer to RAM and consequently reducing computation time. The data access pattern associated with k-blocking is shown in Figure 5. In our stencil code, the time spent loading field data from RAM can be replaced with a shorter loading time from the cache during Laplacian calculations and the computation of the output field when using k-blocking.

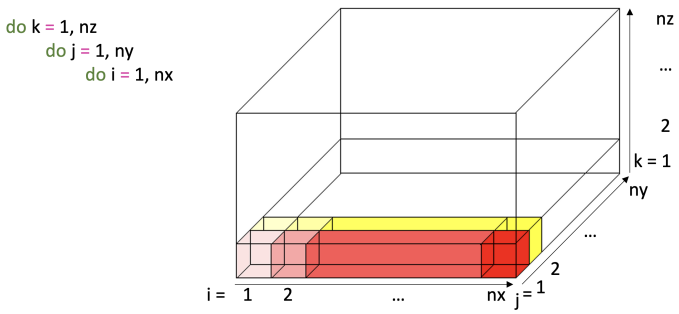

Figure 5. A schematic diagram illustrating the concept of the k-blocking technique.

2.3 Ij-blocking

Ij-blocking also optimizes computational efficiency by improving cache utilization, but different from k-blocking, ij-blocking partitions the data into blocks along x-y planes. The size of each block can be denoted as mx×my×nz. Ij-blocking involves overlap between these blocks, meaning the boundaries of adjacent blocks intersect, and their values are recalculated during the processing of each block. The data accessing pattern of ij-blocking is illustrated in Figure 6. In this study, various combinations of mx and my values are explored across different domain sizes to determine the most effective optimization for the stencil program through the utilization of ij-blocking.

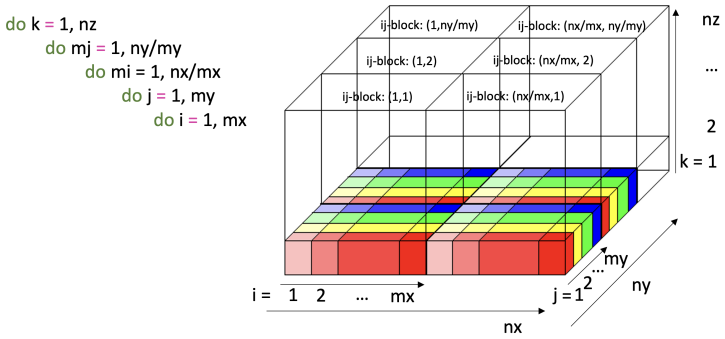

Figure 6. A schematic diagram illustrating the concept of the ij-blocking technique.

3 Experiments

There are several questions that we would like to address regarding ij-blocking:

1. How does the size and shape of the domain affect the runtime per gridpoint and what is the optimal size? What is the optimal size and shape?
2. How does the ij-blocking strategy perform compared to k-blocking strategy?
3. How does the ij-blocking strategy react with varying domain sizes?
4. How does the ij-block size and shape affect the runtime?

To answer the questions above, we have designed several experiments:

1. Simulations with 3 sets of fixed domain sizes (i.e. same number of gridpoints) on the horizontal plane (64 x 64, 512 x 512, 2048 x 2048) with fixed nz=4, but varying the nx and ny. This allows us to investigate the effects of different shapes (i.e., rectangular and square) and the effect of the longer side of the rectangular locating on the i and j directions.
2. Simulations with different sizes of the square domain from 64 x 64 to 8192 x 8192 using different ij-block sizes (by fixing either the i or j direction and increasing the block size until the block size becomes the size of the horizontal plane.
3. Simulations with different shapes and sizes of ij-blocks on different square domains (64 x 64, 512 x 512, 4096 x 4096) to find out the optimal ij-blocks.

4 Results and Discussions

4.1 Experiment 1: Simulation with fixed domain size by varying nx and ny

The experiments are conducted by varying the nx and ny but keeping the target domain size/total grid point in the same layer the same. For example, 64 x 64 is equal to a total gridpoint of 4096, other combinations are 2 x 2048, 4 x 1024, and so on. By running every possible combination with a base of 2, the results of this experiment are shown in Figure 7. The results all show that the closer the domain shape to being a square, the faster the runtime. When it approaches a longer rectangular shape both in i and j directions, the runtime increases. One point to note is that at smaller domain size with total points of 4096, the runtime of rectangular in i direction (i.e. $\text{nx} = 2^{12}, \text{ny} = 2^0$) is lower than the runtime of rectangular in j direction (i.e. $\text{nx} = 2^0, \text{ny} = 2^{12}$). When the domain size becomes larger, there is no significant difference in runtime between rectangular in i direction and rectangular in j direction. The increased runtime in rectangular shape could be due to increased cache misses by loading unnecessary long stride which was not used, while the stencil program requires information in j stride, as shown in Table 1. Despite the slight difference in the extreme end of the rectangular shape, all simulations show that the square shape domain works the most efficiently.

4.2 Experiment 2: Simulation with different square domain sizes using different ij block sizes.

In our initial blocking strategy, we aimed to partition the x-y plane into strips of varying widths, extending from 1 to the full width of the plane, while maintaining a consistent length equivalent to the plane's entire side length. This experiment encompassed 8 different sizes, ranging from 64x64 to 8192x8192. The resulting data is effectively illustrated in the accompanying plot, where the y-axis represents computation time, normalized relative to the quickest time recorded within each individual run. Meanwhile, the x-axis is represented in a logarithmic base-2 scale denoting the change of the block size.

Upon analyzing the plot in the left panel of Figure 8, it becomes evident that there are two discernible clusters of lines. A noticeable trend emerges in the test runs featuring smaller sizes (specifically those with side lengths of 64, 128, and 256): as the block size diminishes, the computation time progressively elongates. Interestingly, the optimal performance within this subset occurs when the block sizes fall within the my range of 2^5 to 2^7.

Conversely, when focusing on the experiments conducted with larger dimensions, an intriguing pattern emerges. Similar to the smaller size runs, the quickest computation times were also observed when the block sizes were approximated within the my range of 2^5 to 2^7. However, a distinctive behaviour emerges beyond this range. As the block size surpasses this range, a sudden surge in computational time becomes apparent. Remarkably, after this threshold is crossed, the computational time stabilizes, demonstrating a relatively consistent value.

4.2.1 Fixed mx, varying my

When changing the parameter, my, which changes the ij-block size with fixed mx, there are 3 distinct groups shown in the left panel of Figure 9. The lowest 3 domain sizes (64, 128, 256) yield the lowest computational time per grid point among other domain sizes. However, when the ij-block sizes are small ($\text{my} = 2^0 - 2^2$), the computation time is 1.2 - 1.8 magnitude larger than their lowest computation sizes. This could be due to the excessive overcomputing of the halo as the ij-block size is too small, and counters the benefits of ij-blocking. When the domain sizes become larger, the effect of overcomputing diminishes as the ij block sizes become larger (as the mx is equal to ny) and the caches are used more efficiently with less overcomputing. The computational time per grid point increases from a group with

a smaller domain to a higher domain. This could be due to the larger overhead in loading the large domain from the memory and the ij-block falls in a different level of cache. The domain of mx=512 shows an interesting feature that its computational time does not change significantly. The domains larger than 512 x 512 have higher computation time when the ij block size is closer to the whole i-j plane. This could mean that when the ij block is too large, it falls out of the cache, and increases cache miss, thus leading to longer computation time.

When looking at the performance of ij-blocking compared to k-blocking in Figure 10, there are improvements in the runtime with the domain sizes larger than or equal to 512 when small ij-block sizes are used. This could be because the ij-blocking in the i-direction avoids the loading of unused data points on the i-stride while following the nature of Fortran which reads 2D data in the i-direction. As the starting size of the ij-blocks is larger than those in smaller domain sizes, the overhead is negligible. The larger the domain size, the more improvement in runtime over k-blocking strategy. At domain sizes smaller than 512, it does not improve its performance while increasing the runtime at smaller ij-block sizes. This can be explained by the excessive over-computing of halo, and inefficient use of the cache as smaller domains usually are expected to stay in the L1 cache. For instance, for a domain size of 256 x 256, a j-stride is 1KB, which fits in the L1 cache.

4.2.2 Fixed my, varying mx

The experiment is similar to the experiment above but the my is fixed and mx is varies. This leads to the rectangular of the ij-block is in j-direction. A smaller ij-block size causes a longer computational time. The computational time converges to the regular k-blocking runtime for all domain sizes. As all the computational time converges in a similar pattern for all domain sizes, this shows that domain sizes are not the constraints. There could be an inefficiency in calling memory in j-direction, as this violates the nature of the way Fortran reads memory.

These experiments show that even with the same ij-block size in the same domain, the direction of ij-block matters. The ij-block pointing in i direction performs better than ij-block pointing in the j direction. In general, ij-blocking has the capability to outperform k-blocking, by choosing a smaller ij-block size for a larger domain. K-blocking works best in smaller domain sizes, while ij-blocking is useful for larger domains when small ij-block sizes pointing in i-direction are chosen.

4.3 Experiment 3: Simulation on different ij-block sizes and shapes on the different domains.

In this experiment, different ij-block sizes and shapes are applied to different domain sizes with fixed nz=4 (i.e. 64 x 64 x 4, 512 x 512 x 4, 4096 x 4096 x 4).

In all three graphs in Figure 11, the left bottom corner shows a similar pattern in higher runtime per grid point. This is expected because when mx and my are equal to 1, each ij-block is programmed to calculate the halo around the ij-block, this leads to over-computing and does not help with improving the computation time.

In the left upper corner of 3 of the graphs, the runtime is slightly slower than the runtime at the right upper corner, which is the runtime of normal k-blocking. At $mx = 2^0$, an increase in my decreases the runtime and then increases again towards the highest my. The decrease in runtime at the first place could be due to the decrease of the effect of overhead and over-computing caused by small ij-block sizes. This effect is also seen when increasing mx along $my = 2^0$. The latter increase in runtime could be due to the ij-blocking violating the nature of Fortran loading data in i-direction. This effect becomes significant when the reduced overhead and over-computing are no longer further improving the runtime.

The resonates with the result in Experiment 2, where it costs extra computation time when the ij-block is pointing towards j-direction.

The increase in runtime at the location around mx = 2^6 and my = 2^{12} of 4096 x 4096 in Figure 11 is unclear to us. This effect becomes more significant with increasing domain size, as this is not observed in the graph of 64 x 64. One possible reason that we could think of is that this case is similar to the case where a rectangular pointing in j-direction causes an increase in runtime. All factors we discussed before contribute to this area which causes uncertainty to us about which factor dominates.

Unlike Experiment 1 where square domain is seen to be the most efficient domain shape, ij-blocking does not follow this pattern as we have to consider the over-computing in small ij-block size. One take-home message in this experiment is that with a larger domain size, a rectangular in i-direction (e.g. large mx, small my) could be beneficial to improving the runtime. This could be based on the fact that the way Fortran load data in i-direction usually dominates over other factors when the size of the data gets larger.

5 Conclusion

To conclude, our experiments have answered our questions at the beginning. Firstly, a square domain is an optimal domain shape no matter what domain size it is. Thus, in planning a model, we could make use of this feature to possibly increase the efficiency of a stencil program. Secondly, ij-blocking performed badly in small ij-block size in small domain size, as there is a significant contribution from overhead and over-computing. Due to time limitations, we could not quantify the actual value of the over-computing time. This could be included in future studies to fully understand the contribution of this issue. In general, we should avoid using small ij-block sizes. In large domain sizes, implementing a small rectangular ij-block in i-direction has shown to be able to outperform k-blocking strategy. Thus, when one is dealing with a large domain size, it is wise to consider implementing ij-blocking in i-direction. One should avoid implementing ij-blocking in j-direction or possibly even in "ik-blocking" as this could increase the runtime, possibly due to inefficiency in loading data as Fortran loads data in i-direction. This could change depending on the programming language used. One interesting future work could be comparing the ij-blocking implementation in different programming languages. Due to the limitation of our knowledge and scope, we could not compare it with other programming languages to truly justify if the inefficiency is actually caused by the specific way of Fortran loads data. Thirdly, the results have shown that different ij-block sizes and shapes can have different runtime in different domain sizes. The pattern gets more complicated in larger domain sizes, as different factors could be magnified and dominate other factors. While we could not fully explain the pattern in larger domain sizes, one take-home message is one should consider more aspects when dealing with larger domain sizes. Despite the simplification of pattern smaller domain sizes, when ij-blocking is implemented correctly with the optimal block size, one can manage to improve the runtime of the stencile program. This could improve the performance as larger domain sizes might be using L2, L3, or even RAM, thus ij-blocking strategy could work if it can reduce the cache miss. This is not the case for smaller domain sizes, as the over-computing elements usually dominate, and k-blocking is the best strategy possibly allowing them to fall within the L1 cache. To conclude, if possible, one should consider using small and square domains with only k-blocking as it yields lower runtime per gridpoint. If a large domain is unavoidable, one should also consider implementing ij-blocking alongside with k-blocking.

References

Fuhrer, O. (2023). *High performance computing for weather and climate.*

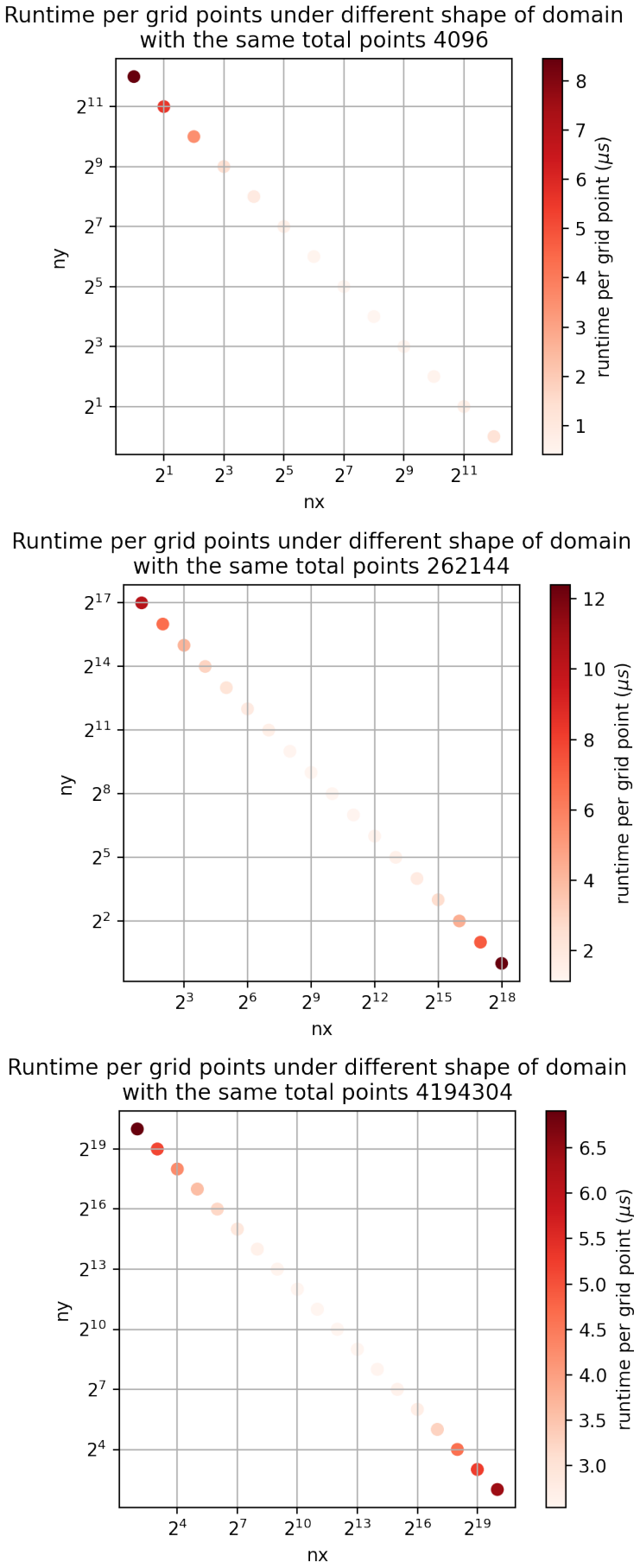

Figure 7. Runtime per grid points given the same total grid points with varying domain shape, namely changing nx, ny.

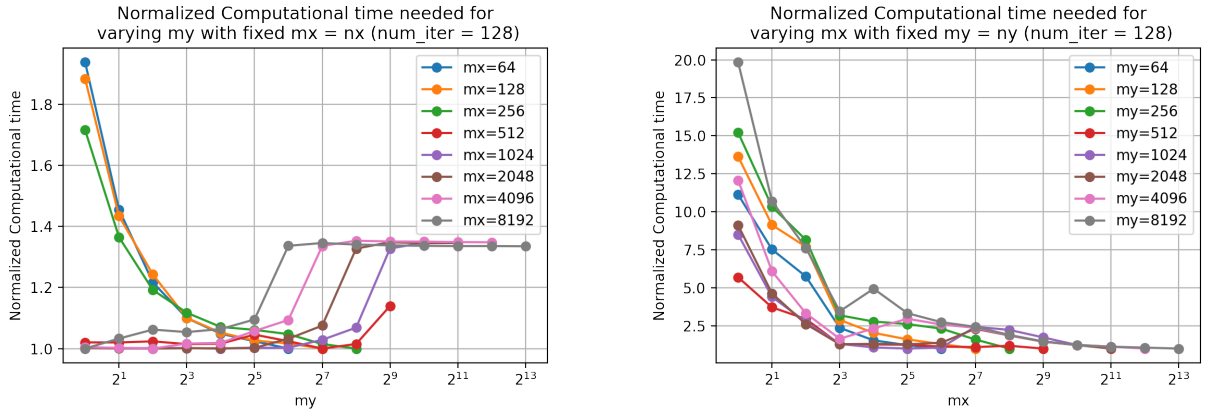

Figure 8. Normalized runtime required for different block sizes. The normalized time is defined by the runtime divided by its minimum runtime obtained in the same fixed block length.

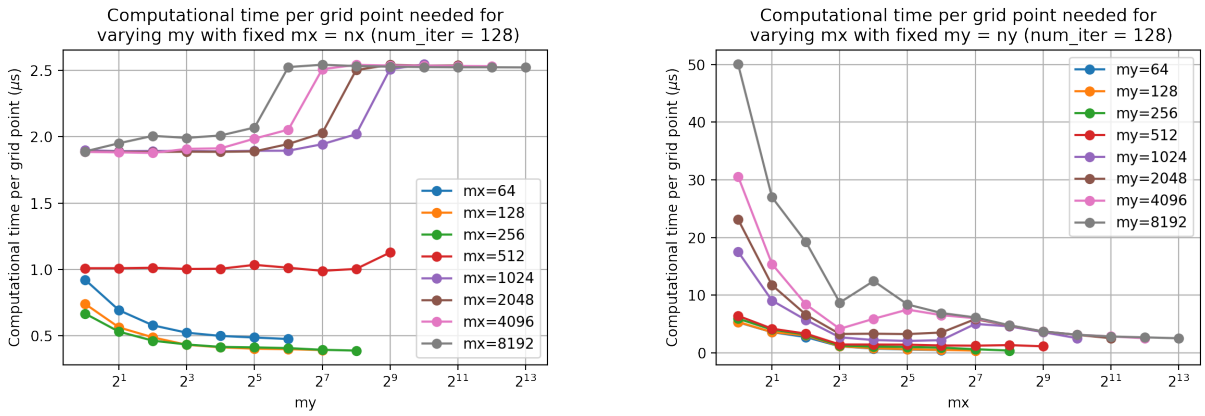

Figure 9. Runtime per grid point required for different block sizes.

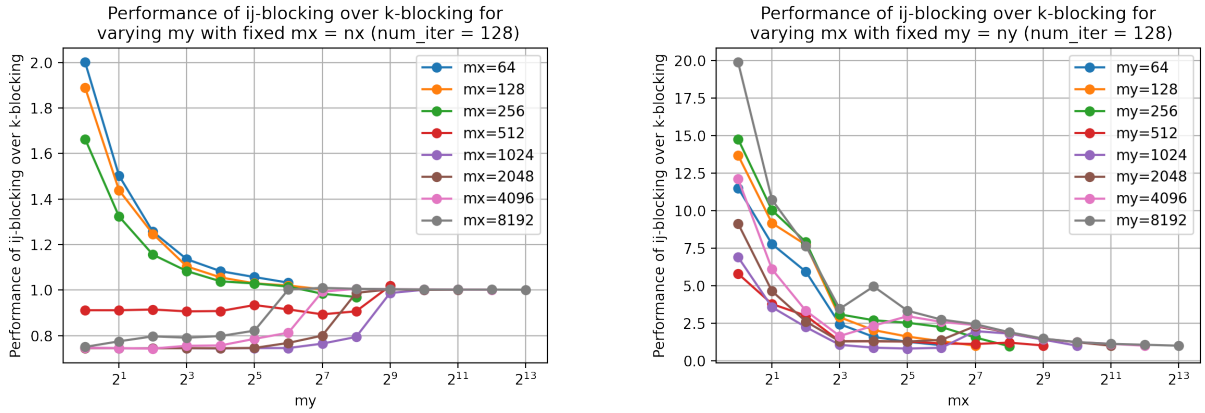

Figure 10. Performance of ij-blocking over k-blocking, examined by the runtime of ij-blocking over that of k-blocking.

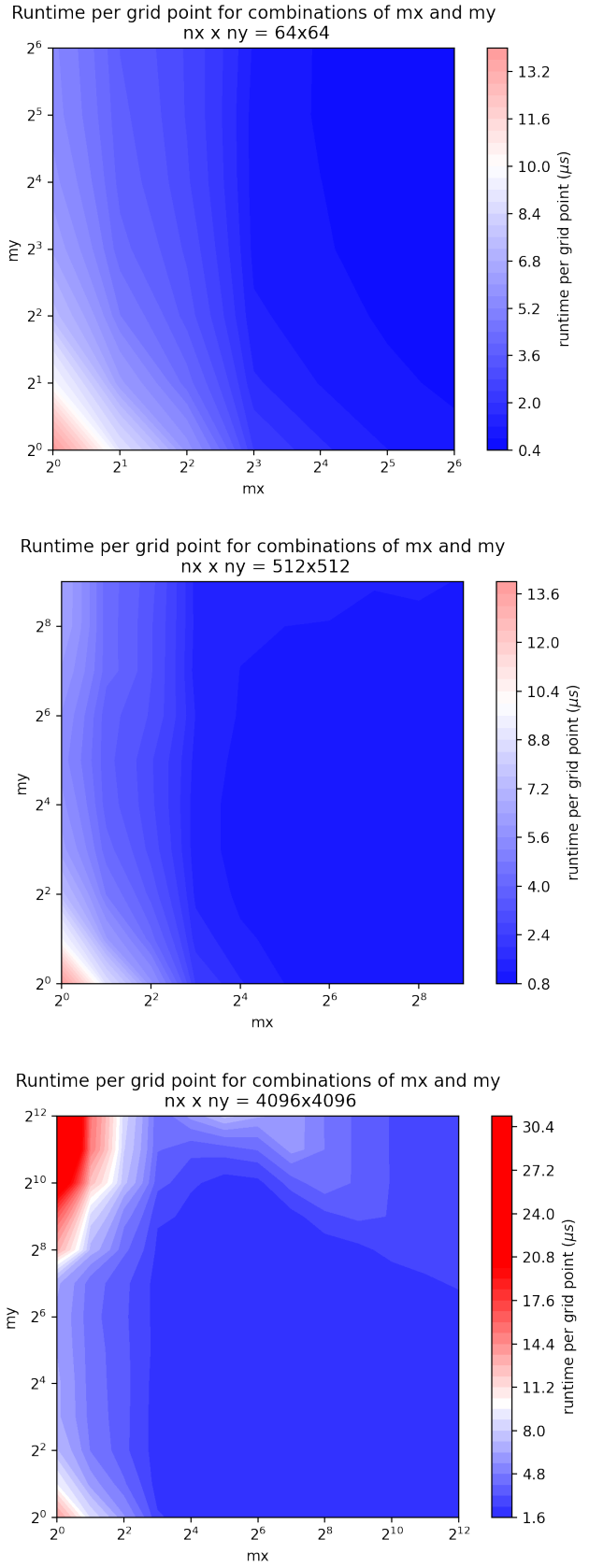

Figure 11. Runtime per grid points given varying block length mx and my.