ETH zürich

IAC Institute for Atmospheric and Climate Science

High-Performance Computing for

Weather and Climate

Spring Semester 2024

Jonathan Melcher, Kwint Delbare,

Lina Bernert, & Nikola Mang

Blocking Strategies for Weather and Climate Models

Zurich, 25 July 2024

Contents

List of Figures

List of Tables

1 Introduction

Partial differential equation (PDE) solvers are critical in scientific applications such as heat diffusion or fluid dynamics. PDE solvers typically use iterative finite-difference techniques that perform nearest-neighbour computations, called stencils, on a time-space grid. Because the data structures involved in these stencil computations are often larger than the capacity of available data caches, memory bandwidth limits the performance of PDE solvers to a small fraction of peak performance, e.g. Kamil et al. [2006].

A cache is a short-access computer memory that stores frequently or recently accessed data for faster retrieval. Modern CPUs have a hierarchy of caches - L1, L2 and L3 - with increasing size and access time; the cache sizes of the CPU used in this study are: 32 KB, 256 KB and 30 MB, with latencies of 4, 12 and 40 clock cycles. However, the cache can only hold a small fraction of a data matrix, so data may have been displaced from the cache by the time it is needed again. This displacement degrades the performance of applications such as PDE solvers that rely on repetitive data access, as the most significant memory latency occurs when data is transferred between RAM and CPU, e.g. Fuhrer [Spring Semester 2024].

To improve cache utilisation at all levels of the hierarchy and increase performance, data locality must be optimised. Locality refers to the tendency of programs to access data and instructions at addresses close to those that have recently accessed. An important algorithmic change to improve data locality is cache blocking. This is an optimisation technique that rearranges data access patterns to pull subsets (blocks) of data into the L1 or L2 cache and operate on these blocks across multiple uses. In this way, the technique reduces the amount of data loaded from RAM, relieves pressure on memory bandwidth and shortens memory latency Fuhrer [Spring Semester 2024]. In addition to spatial blocking, there is also temporal blocking, which attempts to maximise data reuse over multiple iterations, e.g. Lam et al. [1991]. The focus of this study, however, is on spatial blocking.

The effectiveness of cache blocking depends on factors such as data block size, processor cache size and the frequency of data reuse, e.g. Lam et al. [1991]. As a general guideline, block sizes should be approximately one-half to three-quarters of the physical cache size, e.g. CITA [2007]. On Piz Daint, the supercomputer used for our experiments, we can find the optimal block size with double float precision as $32000/(8 \cdot 2) = 2000$ values. In addition, the shape of the block is crucial as all programming languages store 2D arrays with a dominant index and load memory into the cache as a vector of four adjacent values rather than a single value. In Fortran the dominant index is i, thus, tilling in the i-direction loads the fastest Fuhrer [Spring Semester 2024].

For weather and climate models the blocking can be applied in the vertical (k-) and horizontal (ij-) direction: While k-blocking is easy and the least disruptive to our test code, ij-blocking is an interesting addition. Using a stencil program, we aim to determine whether ij-blocking improves the performance of a simple already k-blocked code. In addition, we test the performance impact of the ij-block size and shape and how it behaves for algorithmic motifs of different complexity.

2 Methods

2.1 Stencil program

The stencil program in the test case is a two-dimensional diffusion problem in a three-dimensional space with no interaction between the vertical layers. After generating an initial diffusion field, the stencil program iterates through eight subroutines. The resulting diffusion field is visually validated and its deviation from the initial diffusion field, yield by the k-blocked code, is quantified with the mean absolute error. Unless otherwise stated, we use a stencil with the nearest non-diagonal points (centre Figure 2.1) to calculate the diffusion at point i, j.

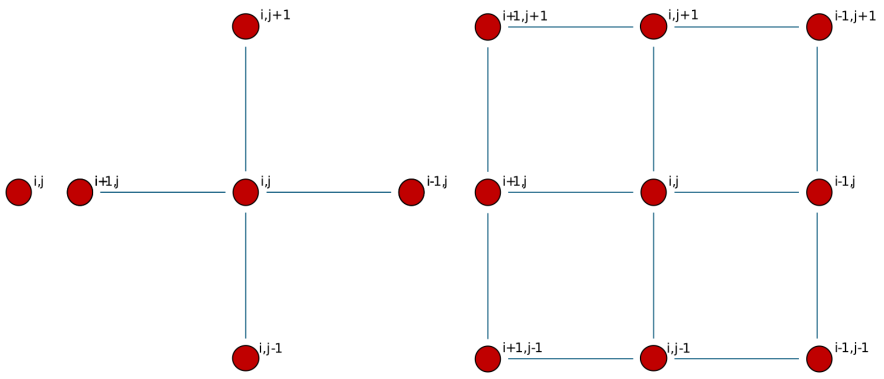

Figure 2.1: Stencils tested for different algorithmic motives. To see the effect of different stencils we test the code with a simple copy operation, a no diagonal stencil (indicated by *-nn*), and a stencil with diagonals (indicated by *-nnn*) (from left to right).

As stencil programming is widely used in weather and climate models, the results from the diffusion problem tested here are a good proxy for how the stencil code would perform for the more complicated Navier-Stokes equation.

2.2 Cache Performance Metrics

In addition to model runtime, the blocking strategies are evaluated by cache performance: The hit ratio indicates the proportion of data requests that are satisfied by the cache without resorting to lower-level memory. The cache utilisation refers to the rate of cache misses or instances where memory is not served by the cache, relative to the total number of requests. Both metrics are derived from CrayPat tracing.

2.3 Optimisation iterations for ij-blocking

Starting with a k-blocked version of the double Laplacian, we test whether additional horizontal blocking improves cache performance and computational efficiency. To do this, we implement ij-blocking in the *apply_diffusion* subroutine in various ways, along with some code optimisations, expecting performance improvements (from top to bottom) for the strategies listed in Table 2.1: First, we naively implemented the blocks by adding two more loops so that we now loop over k, then over the ij-blocks, and finally inside the blocks. This should improve our cache hits as we

Table 2.1: ij-Blocking Strategies in comparison. We expect a performance increase from top to bottom.

Strategy	Description
ij-blocking2	Dividing the horizontal plane into smaller blocks.
ij-blocking inline	Inlining by replacing the call to the Laplacian with the operation itself.
ij-blocking small	Save only a small block of the first Laplacian field.
ij-blocking math	Pass the size of the for-loop to the compiler.
ij-blocking comp	Replace runtime with compile variables.

now hold the neighbouring points in memory for the next stencil computation. Next, we replaced the double Laplacian function call with an inline statement. This should improve performance as the compiler now can more readily optimize the machine code. To further optimise cache usage, we demoted the temporary field from a 3D field to a 2D field, of the same size as the ij-blocks. Finally, we aimed to assist the compiler by minimising the number of runtime-determined loops: This involved making the inner loops iterate over the smaller temporary field instead of the input and output fields. In addition, we tried to make the domain size and ij-block sizes compile-time variables. This approach should allow the compiler to eliminate control structures and use other optimisation strategies.

To test the performance improvements we apply these optimisations to a square domain of varying sizes, ranging from 64 to 2048, increasing in powers of two. The following variables are held constant: 16 vertical layers, 256 iterations, and an ij-block size of 32×32. To make the different runs comparable, the runtime is normalised by the k-block runtime.

2.4 Optimal ij-block size and shape

After evaluating the effect of different blocking strategies, we use the best-performing blocking strategy (*ij-blocking math*) to examine how varying the size and shape of the ij-blocks affects the runtime. As outlined in Section 1, we expect that a temporary field of about 2000 values and a rectangle skewed in the i-direction would give the best performance. We therefore run the code with a constant grid size of 4096×4096, 16 vertical levels and 256 iterations, testing all possible combinations of $X \times Y$ with $X,\ Y \in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$ for the ij-block shape. To determine the optimal block size, we plot the normalised runtime (relative to the k-block runtime) against the size of the ij-block and colour code the shape of the rectangle (i- versus j-skewness). To examine how performance varies with block shape, we analysed the runtime of rectangular ij-blocks of varying shapes and sizes, normalised to the runtime of a square of the same size.

2.5 Chaning algorithmic motif and stencil definition

Finally, we gradually increase the complexity of the diffusion operator approximations (cf. Table 2.2) and compare their runtime for the *k-blocking* and *ij-blocking math*. All diffusion operators use the same amount of memory ($nx,\ ny,\ nz\ =\ 128,\ 128,\ 64$ and 128 iterations) but perform a different number of floating-point operations (FLOPs), thereby increasing accuracy.

Table 2.2: Diffusion schemes of different complexity. The different algorithmic motifs were applied to a grid size of $1024 \times 1024 \times 64$.

Motif	Description	# FLOPs
copy	Simple copy stencil.	
averaging	Averaging over neighbouring values (= 0th order diffusion).	4
lap	1 × Laplacian (2nd order diffusion).	7
kblocking	2 × Laplacian (4th order diffusion) with k-blocking.	12
ijblocking-math	2 × Laplacian (4th order diffusion) with ij-blocking.	12

The computational overhead of ij-blocking is determined by comparing the performance of the simple copy stencil program (left Figure 2.1) for the k- and ij-blocked versions.

To understand the impact of the stencil definition, we also run the 0th and 2nd order diffusion experiments using an alternative stencil program that includes the next nearest non-diagonal neighbours ($(i+1, j+1), (i+1, j-1)$ etc., right Figure 2.1). Since the next-nearest-neighbours stencil requires more values for each calculation, the number of FLOPs per diffusion operator increases (# FLOPs for *averaging*: 8 and *lap*: 11).

3 Results and Discussion

3.1 Performance impact of the blocking strategy

As shown in Figure 3.1, dividing the horizontal plane into smaller horizontal ij-blocks (*ij-blocking2*) and inlining the Laplacian (*ij-blocking inline*) are slower than the original k-blocked code due to the additional computational overhead. Making the inner loops iterate over a smaller temporary field instead of the full in- and output fields (*ij-blocking small*) minimises the number of runtime-determined loops and reduces the runtime slightly compared to *k-blocking*. A performance improvement is observed for *ij-blocking math*, because in this blocking strategy the compiler can optimise the loops due to fixed iteration bounds on the loop variables. Compared to the original k-blocked code, the improved *ij-blocking math* gives an almost 25% speed up.

Overall, the impact of the blocking strategies on the normalised runtime (concerning *k-blocking*) varies with the domain size. For the limited number of experiments we did, introducing compile-time variables (*ij-blocking compile*, not shown) instead of run-time variables shows variable performance: it is slower for smaller k-dimensions, but faster for k-dimensions of 64 or more. Thus, the choice between *ij-blocking math* and *ij-blocking compile* as a blocking strategy should be context-specific, but needs to be further tested. Although we cannot exactly explain the size dependency of the *ij-blocking compile* strategy, we would expect it to be faster due to the elimination of control structures and compile-time optimisations.
In addition to the runtime, we also analyse cache performance for *ij-blocking-math* and *k-blocking* (cf. Table 3.1): Here the ij-blocking technique demonstrates superior L1 cache performance over k-blocking by maximising cache hit ratios and utilisation through optimised data locality, thereby reducing slower main memory accesses. This is further highlighted by the significantly lower L2 to L1 bandwidth for ij-blocking compared to k-blocking, indicating less frequent data transfers

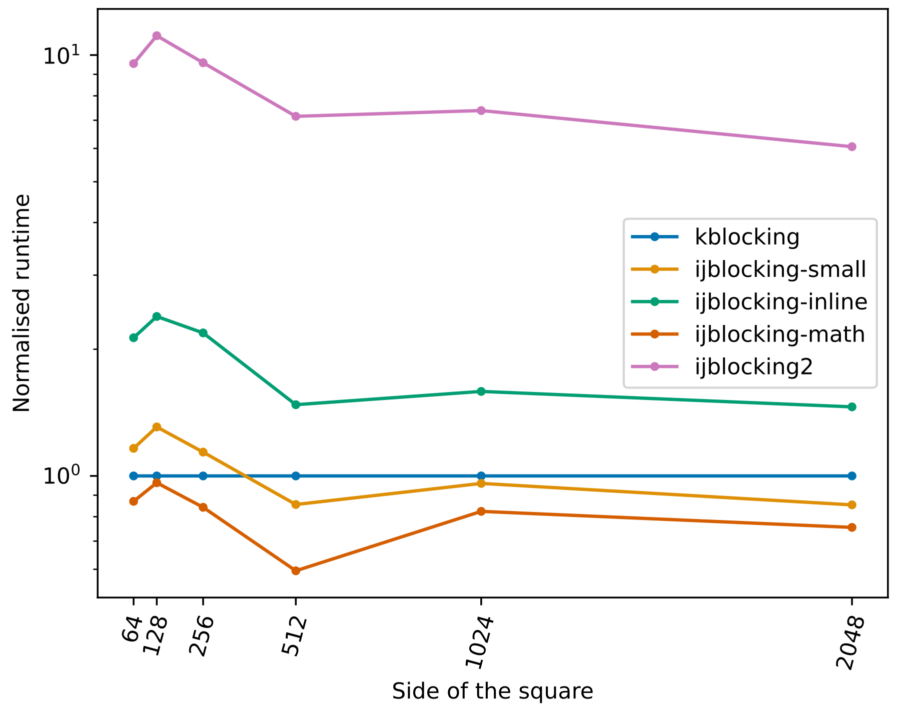

Figure 3.1: Comparison of the runtime of the different ij-blocking strategies for the double Laplacian code. All blocking strategies were applied to a domain size of varying size from 64×64 to 2048×2048 with 16 vertical layers 256 iterations and an ij-block size of 32×32. The runtime was normalized against the k-blocking performance to compare with the baseline.

between caches and more efficient data flow. These improvements can also be seen in the lower average time per call. Also, considering that the CrayPat measured overhead is 0.0% for both, the difference in time per call is trustworthy.

Table 3.1: Cache performance for k- and ij-blocking-math. The cache performance of the different blocking strategies was tested on a $4096 \times 4096 \times 16$ grid. A block size of 64×64 was used for the ijblocking.

Cache Metric	*k-blocking*	*ij-blocking-math*
L1 cache hit ratio	68.4%	81.9%
L1 cache utilisation	0.40 avg hits	0.69 avg hits
L2 cache hit ratio	39.3%	21.7%
L1+L2 cache hit ratio	80.8%	85.9%
L1+L2 cache utilisation	0.65 avg hits	0.88 avg hits
L2 to L1 bandwidth	14.072 GiB/sec	9.313 GiB/sec
Average Time per Call	0.462666 sec	0.381321 sec
CrayPat Overhead (Time)	0.0%	0.0%

It is worth noting that L1 cache performance is more efficient than L2 cache performance for both blocking strategies. By accessing more data directly from the L1 cache, the need to fetch data from L2 or lower-level memory is minimised, improving overall computational efficiency. Given that the L1 cache hit ratio with *ij-blocking-math* is close to 100%, further cache performance improvements with other blocking strategies are unlikely unless the performance of the lower-level cache is enhanced. A more optimal block size might, however, increase the cache performance significantly.

3.2 Experiments with ij-block size and shape

Comparing the runtime of different ij-block shapes and sizes (cf. Figure 3.2), it can be seen that blocks skewed towards the i-direction (indicated in red) perform better, with an optimal range between a ij-block size of 10^3 and 10^4. As expected (cf. Section 1, Section 2), the fastest runtime is achieved for an ij-block with dimensions 256×8, i.e. a block size of 2048, skewed maximally in the i-direction.

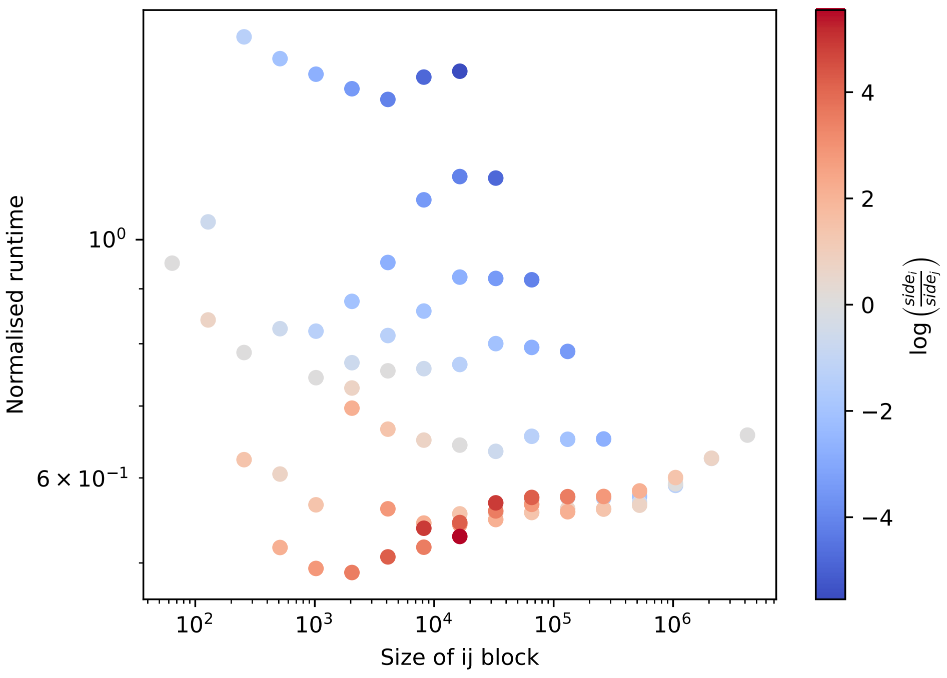

Figure 3.2: Correlation between runtime and size of the ij-block. Simulations spanning all combinations of rectangles taken from $\{8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$, with 16 vertical levels, 256 timesteps, and a domain size of 4096×4096. Runtime was normalised against k-blocking performance as a baseline. There is no significant correlation between ij-block size and runtime, which seems to be more influenced by block shape.

Examining the skew of the ij-blocks in Figure 3.3, we observe that when the ratio $r = \frac{\text{size}_i}{\text{size}_j} < 1$, i.e. when the ij-blocks are skewed in j-direction, the code runs slower compared to the square setup, and it runs faster when $r > 1$. This tendency is due to Fortran's i-major behaviour. In addition, smaller block sizes (16×16 - 64×64) seem to benefit more from strong skewing of the block in i-direction. The small speedup for the block sizes 128×128 and 256×256 could be explained, by the temporary field already fitting into the L2 cache.
In order to arrive at an exact optimal ratio and size, estimates of the uncertainty are required. Anecdotal evidence suggests that the fastest runtimes that occur for block sizes of 64×64 and 32×32 (see Figure 3.3), as well as the three fastest configurations in Figure 3.2, may be within the uncertainty range of our experiment. Therefore, further trials of the same experiment are needed to draw a confident conclusion. What is clear is that the i-major nature of Fortran provides a significant advantage for the no-diagonal stencil double Laplacian code when using a rectangle that is skewed - potentially as much as possible - in the i-direction.

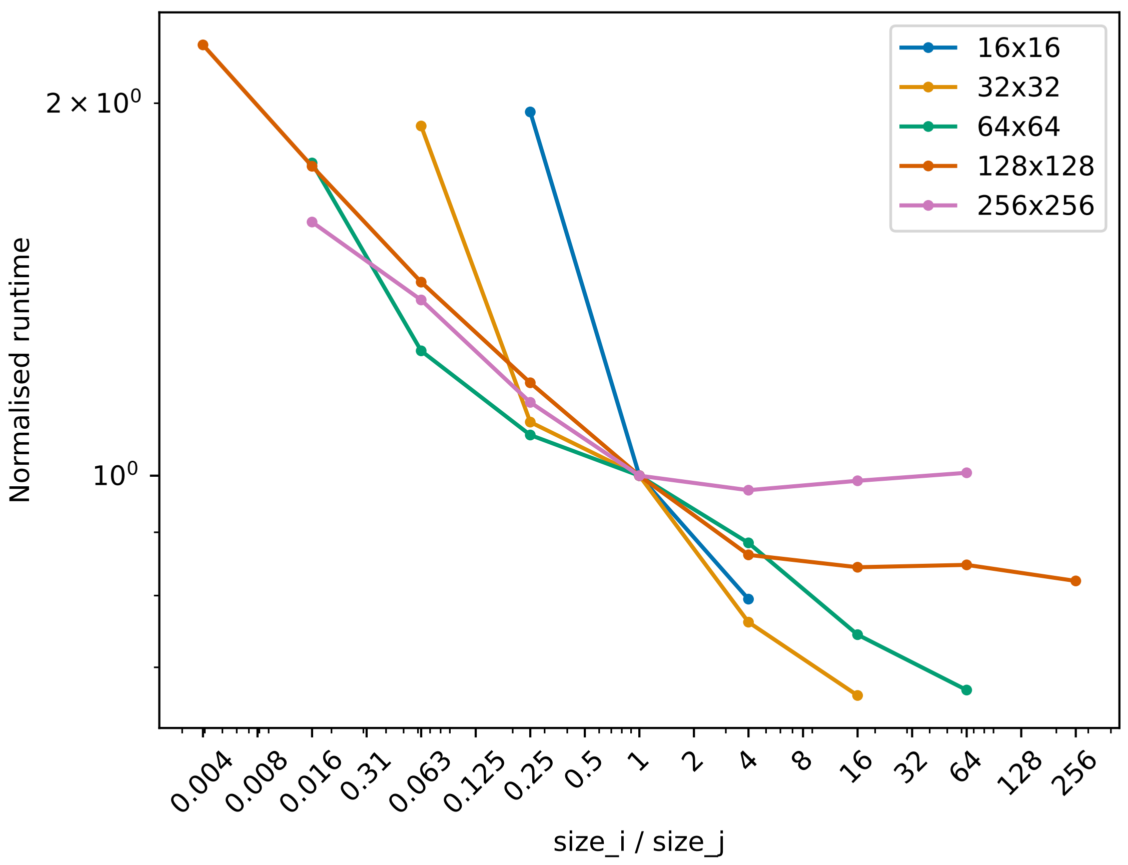

Figure 3.3: Comparison of shape of the ij-block. The simulations performed for Figure 3.2 are normalised using a square of the same size as the ij-block. The green data point on the bottom right represents the runtime of the 512×8 block compared to the runtime of the 64×64 ij-block. All points pass through $(1, 1)$, where the square is compared to itself.

3.3 Effects of the algorithmic motif and stencil definition

A runtime comparison between the ij-blocked and k-blocked versions of different algorithms, as shown in Figure 3.4, reveals that ij-blocked algorithms consistently outperform their k-blocked counterparts due to superior cache usage. However, it's important to note that the k-blocked version is not optimised, so the comparison is not entirely ”fair”. The runtime differences would probably decrease if similar optimisations were applied to the k-blocked version.

The runtime difference becomes particularly large for more complex algorithms: while the runtime for the *ij-blocking math* code increases only slightly with higher FLOP counts, the runtime for the *k-blocking* version rises significantly. Thus, optimising spatial cache usage with ij-blocking becomes increasingly beneficial as algorithm complexity increases and memory transfer becomes a limiting factor. The memory bandwidth constraint also explains why computationally intensive algorithms with *ij-blocking math* have a similar runtime to simpler ones: the additional floating point operations are performed while the program without ij-blocking would otherwise be idle, waiting for memory access. This allows greater accuracy without the runtime penalty of the k-blocked version.

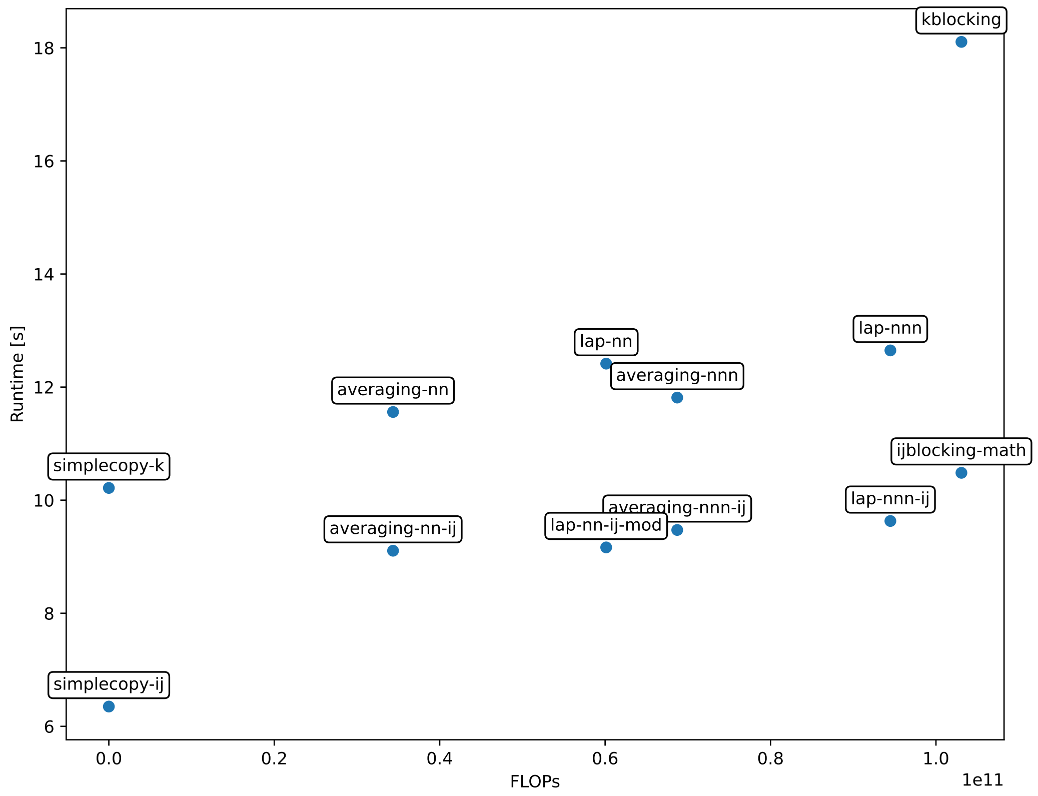

Figure 3.4: Runtime comparison for different algorithms. Each algorithm listed in Table 2.2 is compared for the ij-blocked version (with -ij label) and the k-blocked version. (no extra label). Number of FLOPs is used as an indicator of algorithmic complexity.

Interestingly, the *ij-blocking math* version of the simple copy stencil shows a speedup of about 34% (averaged over a sample of 100 repetitions) in runtime compared to the k-blocked version, suggesting that ij-blocking does not introduce any additional computational overhead compared to k-blocking. However, much of this performance gain is probably due to the lack of optimisation in the k-blocked code. From the speed-up seen when applying *ij-blocking math* in the simple copy algortihm, the ij-blocking seems not to introduce the computational overhead is estimated to be about 34% of runtime for the k-blocked algorithm.

Contrary to expectations, the increase in runtime from algorithms that also consider diagonal neighbours to those that only use non-diagonal neighbours is minimal, despite the higher number of FLOPs. This could be explained by the fact that in this experiment the shape and size of the ij-blocks are optimised for cache usage, so loading additional values is highly efficient.
Comparing the averaging algorithms with the once-applied Laplacian shows that both neighbourhoods are faster for the averaging algorithms. The temporary field which is used in the Laplacian, but not the averaging one, is a likely cause for this difference in runtime since it needs another memory write and read operation.

4 Conclusion

In conclusion, the study shows that ij-blocking can significantly improve the performance of simple k-blocked code by using a smaller temporary field, reducing the number of runtime-determined loops, and allowing the compiler to optimise the loop via fixed iteration bounds. Thus, with our best blocking strategy, we achieve a speedup of almost 25% compared to the original k-blocked code.

In addition, the size and shape of the ij-block have a significant impact on the runtime. The fastest runtime is observed with a 256×8 ij-block because the i-skewed shape can exploit Fortran's i-major behaviour and the 2048 block size aligns with the physical cache size of the Piz Daint supercomputer.

ij-blocking consistently outperforms k-blocking across a range of algorithmic motifs, especially for more complex algorithms where the runtime increase with ij-blocking is relatively small. A comparison of the runtime of a simple copy algorithm suggests that ij-blocking does not add any computational overhead over k-blocking, although this is likely to be largely due to a lack of optimisation of the k-blocked code. The runtime increase introduced by using a stencil that also considers the diagonal neighbours is small.

5 Data and Code Availability and Computer resource usage

The code for this project can be found here github. All experiments are executed on Piz Daint, a Cray XC40/XC50 supercomputer. For technical details see CSCS [2024].

References

CITA. Applying optimization strategies - cache blocking, 2007. URL https://www.cita.utoronto.ca/~merz/intel_c10b/main_cls/mergedProjects/optaps_cls/common/optaps_perf_optstrats.htm#:~:text=As%20a%20general%20rule%2C%20cache,half%20the%20physical%20cache%20size. Accessed: 2024-07-25.

CSCS. Piz daint, 2024. URL https://www.cscs.ch/computers/piz-daint. Accessed: 2024-07-25.

O. Fuhrer. High performance computing for weather and climate. *ETH Zurich*, Spring Semester 2024.

S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, 2006.

M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.