CACHE HIERARCHY AND LOOP OPTIMIZATIONS

Report

Basil Ruch[*]
ETH Zurich, Switzerland
Department of Mathematics

David Strassmann[†]
ETH Zurich, Switzerland
Department of Mathematics

September 11, 2023

Abstract

The effective runtime of an algorithm on a computer is determined by many hardware and software factors. Despite these complex relationships, cache-awareness is crucial for high performance in various applications and on several computational devices. In this work, we focus on single-core CPU cache optimizations for stencil computations. Our self-written cache simulator shows that changes in runtime due to blocking can often be explained with a simulated cache hit rate, and that temporal blocking can be highly beneficial for certain stencils. Yet, the simulator can only capture different access patterns and is agnostic about other important effects such as instruction-level parallelism (ILP) and vectorization.

Introduction

Fast code is extremely hard to obtain. Once a task meant to be solved on a computer is well-defined, choices in the relevant components such as the algorithm itself, the software, the compiler, the microarchitecture and even the code style all have an effect on the final runtime. Unfortunately, without significant effort from programmers these components usually fail to work together ideally and the resulting code is far from optimal. In this work, we focus on an optimization technique called blocking, which is applied directly in the source code to increase the effectiveness of a certain hardware component, the so-called cache. We apply this technique upon stencils, which belong a category of computations where locality is inherently crucial.

[*]ruchb@student.ethz.ch
[†]dstrassmann@student.ethz.ch

```
// copy stencil
b(i,j,k) = a(i,j,k) // (1)

// stencils in x-direction
b(i,j,k) = a(i,j,k) + a(i-1,j,k) // (2)
b(i,j,k) = a(i+1,j,k) + a(i,j,k) + a(i-1,j,k) // (3)

// stencils in y-direction
b(i,j,k) = a(i,j,k) + a(i,j-1,k) // (4)
b(i,j,k) = a(i,j+1,k) + a(i,j,k) + a(i,j-1,k) // (5)

// 2D stencil
b(i,j,k) = a(i,j,k) + a(i-1,j,k) + a(i+1,j,k) + a(i,j-1,k) +
           a(i,j+1,k) // (6)

// 3D stencil
b(i,j,k) = a(i,j,k) + a(i-1,j,k) + a(i+1,j,k) + a(i,j-1,k) +
           a(i,j+1,k) + a(i,j,k-1) + a(i,j,k+1) // (7)
```

Listing 1: All considered stencils expressed in three-dimensional arrays.

In the following, we give a proper introduction to stencils and CPU caches. Then, we present our cache simulator which we use to measure hit rates, followed by various experiments concerning spatial and temporal blocking. Finally, we provide a concise conclusion and possible extensions to this work. The appendix deals with reproducibility and contains additional results.

Stencils

A stencil refers to a computational pattern where discrete values on a grid are updated through a weighted sum of possibly itself and its neighboring grid points. This pattern is very common in scientific applications such as numerical solvers for partial differential equations (PDEs) [1] or image processing [2], and often dominates the overall runtime as part of larger systems [3].

Listing 1 shows our choice of stencils expressed in pseudocode with three-dimensional arrays. From now on, we refer to the first index from left to right as the x-dimension, the second as the y-dimension and the third index as the z-dimension. As higher-dimensional stencils often require non-uniform memory accesses, it is crucial for the runtime to keep the number of jumps in memory as low as possible.

Cache

Depending on the operational intensity, algorithms can either be compute-bound or memory-bound. In the compute-bound case, the performance is limited by the number of floating-point operations (FLOPS) per time unit. For the memory-bound case however, which most algorithms belong to, the data transfer between memory and the computation units is the limiting factor.

```
cache.access(&b(i,j,k));
cache.access(&a(i,j,k));
cache.access(&a(i-1,j,k));
cache.access(&a(i+1,j,k));
cache.access(&a(i,j-1,k));
cache.access(&a(i,j+1,k));

b(i,j,k) = a(i,j,k) + a(i-1,j,k) + a(i+1,j,k) + a(i,j-1,k) +
           a(i,j+1,k);
```

Listing 2: Example of the usage of the `Cache` class with the 2D stencil.

This phenomenon is prevalent in computer systems and often referred to as the Von Neumann bottleneck. In the case of CPUs, caches on the chip try to mitigate this problem by temporarily storing recently fetched data for later use. Many fairly recent CPUs have three levels of cache, L1, L2 and L3 with increasing size and decreasing bandwidth, which is an incomplete [4] but sufficient memory model for our purposes.

This caching approach is only beneficial when a given algorithm has spatial or temporal locality, which is the norm for most applications. As stencils are clearly memory-bound, this means that in order to increase performance we need to restructure memory accesses such that frequently used data stays in cache long enough until it is reused.

Cache simulator

There are several metrics to measure the performance of an algorithm. An obvious choice is to measure the time it takes to run the algorithm on a specific system. This triggers some drawbacks since this measurement depends on the system that runs this algorithm. Additionally, other programs which run on the same system may interfere with the measurements, and the compiler optimizations can also make a comparison difficult. To minimize the influence of other programs running in parallel the measurements are usually repeated several times. Yet, the main issue with the compiler optimizations still remains: If we had two versions of an algorithm, we would have to look at the CPU instructions to evaluate if the compiler applied comparable optimizations to both programs. Another performance metric is the percentage of memory accesses which can be fetched from the cache instead of the main memory. This is the so-called hit rate. Due to the impact on performance, a normal system usually does not record the accesses to memory and which of them can be processed by cache or by main memory.

In order to compare different versions of a stencil operating on a field by their hit rates, a cache simulator is implemented. An instance of the `Cache` class can simulate a cache with parameters `S` (number of sets), `E` (associativity), `B` (cache block size in bytes) and `m` (address bits). Due to alignment reasons the selection of `S`, `E` and `B` as powers of two becomes clear. The intention of this class is to call the `access(address)` member function on every memory access the

algorithm performs. See listing 2 for an example.

The functionality of the `Cache` class relies on the model described in [5]. We have implemented the two replacement policies least recently used (LRU) and least frequently used (LFU). Further crucial functions of the `Cache` class are the following:

- `get_hit_rate()`: returns $\frac{\#\text{HITS}}{\#\text{ACCESSES}}$.
- `get_cold_miss_rate()`: returns $\frac{\#\text{COLD\_MISSES}}{\#\text{MISSES}}$. A cold miss is a miss if the cache block has not been evicted before. Assuming no warm-up those misses are significant and can not be omitted.
- `reset_statistics()`: sets all counters to zero without touching the actual cache memory.
- `flush()`: flushes the cache and resets the statistics.

Two-level cache

In order to find a good trade-off between the cache size and the latency of the memory fetch there are typically multiple cache levels in a system. The closer the cache is located to the CPU the lower the cache size and the faster its data can be loaded into the registers. The measurements are performed on a system with the cache configuration in table 1.

Cache level	Configuration
L1	32 768 bytes, `S`= 64, `E`= 8, `B`= 64 bytes
L2	262 144 bytes, `S`= 1024, `E`= 4, `B`= 64 bytes
L3	16 777 216 bytes, `S`= 16 384, `E`= 16, `B`= 64 bytes

Table 1: Cache configuration of the system in use for the experiments.

In order to simulate parts of this configuration, a class `TwoLevelCache` is implemented, which could easily be extended to more cache levels. The two level cache implements a write-through policy so that the data in L1 is a subset of the data in L2. Therefore, the cache simulator enables to record all memory accesses and determines if the access was a hit in L1, L2 or neither of both. In case of a miss the simulator can even determine the type of the miss, i.e. whether or not it is a cold miss.

Experiments

In this section, we present several numerical experiments related to the stencils in listing 1. The implementation is done in C++ with three-dimensional Eigen [6] tensors and the resulting executable is run on a single CPU core. Eigen is a C++ header-only library for linear algebra with elegant functionality and versatile support. Note that vectorization is explicitly disabled

```
for (size_t k = halo_nz; k < nz - halo_nz; ++k)
  for (size_t j = halo_ny; j < ny - halo_ny; ++j)
    for (size_t i = halo_nx; i < nx - halo_nx; ++i)
      b(i,j,k) = a(i,j,k);
```

Listing 3: Basic implementation of the copy stencil in C++.

```
for (size_t k = halo_nz; k < nz - halo_nz; ++k)
  for (size_t j = halo_ny; j < ny - halo_ny; j += block_ny)
    for (size_t i = halo_nx; i < nx - halo_nx; i += block_nx)
      for (size_t l = j; l < j + block_ny; ++l)
        for (size_t m = i; m < i + block_nx; ++m)
          b(m,l,k) = a(m,l,k);
```

Listing 4: Two-fold blocked version the copy stencil in C++.

through compiler flags as we try to focus on the memory fetching effects of caching alone. For more details on the implementation, see the reproducibility section in the appendix.

All stencils are implemented in the most cache-friendly way. As Eigen tensors store elements in column-major order by default, this means that we always loop through the leftmost index first and then incrementally move to the right, which matches the storage order in memory. For stencil (1) in listing 1, a basic implementation is shown in listing 3. We take the opportunity to introduce halo variables for the loop bounds which denote the number of halo points in each dimension. In the case of PDE solvers, the boundary conditions decide what happens in this halo region. For our purposes, we simply define a constant halo size and leave the halo values untouched.

A common optimization technique entails dividing potentially large memory chunks into smaller blocks, which is called loop blocking. These chunks then hopefully use the available caches more efficiently and reduce the number of direct memory fetches. It is clear that the size of the chunks is crucial. If the block size is too small, then we may suffer from loop overhead, and we miss out on performance. On the other hand, if the block size is too large, then the cache contents may be frequently replaced and blocking has no effect. Ideally, the working set size of a block fits into the L1 cache. Listing 4 shows a sample implementation for two-dimensional blocking in x- and y-direction for the simple copy stencil in listing 1, where `block_nx` and `block_ny` denote the block sizes.

Symmetric spatial blocking

Figure 1 shows the runtime per grid point and the simulated L1 and L2 hit rates for stencil (7) in listing 1 with the straightforward implementation serving as a baseline and a blocked version with block size 16×16. This block size guarantees that the blocks fit in L1 cache.

A full description of the parameter space can be found in the appendix. A working set size having multiple markers means that this specific working set can be achieved through different combinations of the x- and y-dimension. Both plots in figure 1 contradict the expectation that blocking decreases the runtime or increases the hit rate. The figure even shows that the version without blocking performs better at higher working set sizes, both in terms of runtime and L1 cache hit rate. For lower working set sizes the runtime and the hit rates are comparable to the implementations with and without spatial blocking, respectively. One can also see that there is some variability in the runtime for constant working set sizes due to the change in domain aspect ratio. Additionally, considering the runtime a smaller working set is preferred as expected. Finally, we observe that a working set size smaller than the cache size achieves a perfect hit rate of 1 for both the L1 and L2 cache.

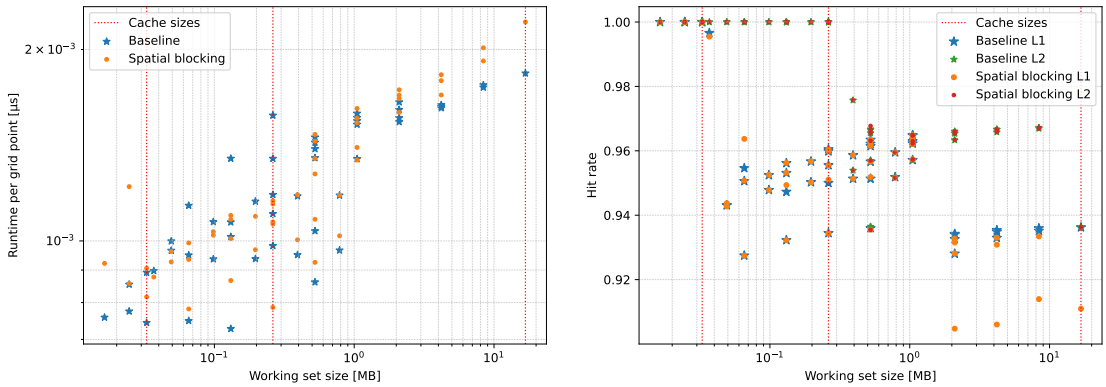

Figure 1: Stencil (7) runtime per grid point (left) and cache hit rates (right) for variable working set size.

An explanation that blocking does not outperform the baseline implementation could be that such a small stencil has relatively low locality. This means that for evaluating a specific element $b(i, j, k)$ only the two neighbors in each dimension are needed. Comparing this with an operation with high locality like the matrix multiplication, we do not use full columns and rows and therefore blocking does not show any performance improvement. One could argue that this effect becomes less apparent with larger stencils, i.e. blocking may then lead to faster runtimes.

Asymmetric spatial blocking

There are two hypotheses underlying the experiment in this section. Blocking in x- and y-direction for the stencil (6) is not equally effective, and the cache hit rate is a good measure for performance. Due to the parameter space, the largest block sizes effectively mean no blocking since the tensor dimensions exactly match the dimension of the block size. Note that as always, there is no blocking in z-direction.

The results are plotted in figure 2. The cache hit rate as a metric for performance definitely makes sense since the time the algorithm needs to complete is positively correlated with the hit rates. Analyzing this plot further, we can see that a blocking of 128 elements in x-direction,

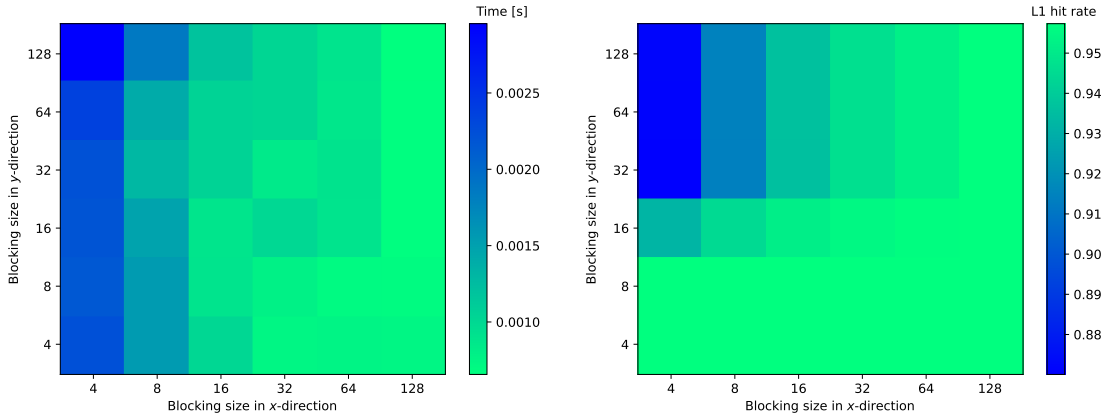

Figure 2: Measured time (left) and recorded hit rates (right) by varying the block size in x- and y-direction.

```
for (size_t n = 0; n < N; n++) {
  stencil_2D(a, b, halo_nx, halo_ny, halo_nz);
  b = a - alpha*b;
  if (n < N - 1)
    std::swap(a, b);
}
```

Listing 5: Baseline implementation of the time step in C++.

which is equivalent to no blocking, outperforms both the runtime and the hit rate of all other choices. Things are different in y-direction since decreasing the block size increases the hit rate and displays a slight tendency to decreases the runtime.

Compared to matrix multiplication, blocking for the 2D stencil has less of an impact on the performance. We assume that the reason of this observation is caused by the allegedly low locality of the 2D stencil. Again, the impact would presumably increase if the stencil had larger dimensions such that one stencil operation does not fit into the L1 cache.

Temporal blocking

According to the results of the asymmetric blocking experiment, spatial blocking does not have the desired beneficial effects. Another approach is to block in time, which is referred to as temporal blocking. Thus, the hypothesis of the next experiment is that temporal blocking has a positive impact on the hit rate. As observed in the previous section, the hit rate is negatively correlated with the runtime. Therefore, in this experiment we only consider the hit rates. As a baseline, the implementation in listing 5 is used, where the function `stencil_2D()` applies the 2D stencil without spatial blocking.

Figure 3 shows the temporal blocking approach. Since all tensors lay in memory in column-

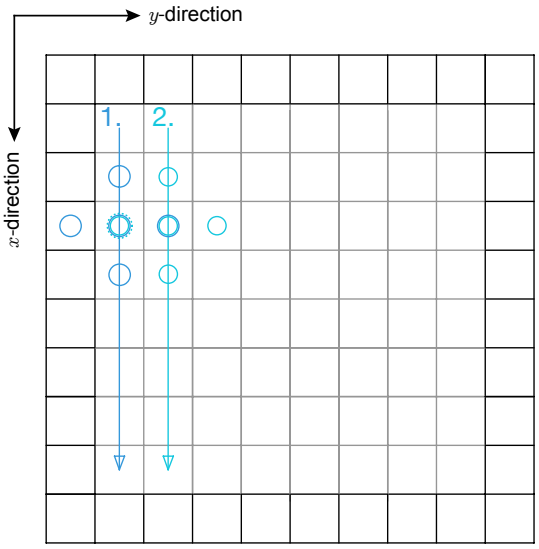

Figure 3: Temporal blocking algorithm. The innermost loop iterates in x-direction and from the second y-iteration onwards the algorithm does the time step in the previous column (dotted circle). The colors correspond to the iterations in y-direction. Black squares denote the halo elements, and the gray ones denote the interior domain.

major order, the stencil is applied in the innermost loop in the first dimension, which is the x-direction. Regarding the y-iteration, once the innermost loop is done the algorithm performs the time step on the previous column. This could potentially be more sophisticated if the algorithm locally alternates between stencil application and time stepping which would result in a staircase-like update form.

The recorded hit rates are displayed in figure 4. First and foremost, it shows that with a small working set size fitting in L1 there are no differences between the baseline and the temporal blocking implementation, as with a cache warm-up every access can be fetched from L1 cache. If the working set size exceeds the L1 cache, the plot shows that the temporal blocking has an increased hit rate of approximately 30% compared to the baseline implementation. The right plot exhibits the same phenomena in the L2 cache.

Doing the time step right after applying the stencil increases the locality of the of temporal blocking and this most likely also improves the hit rate. Another interesting finding is that the equality of the dimensions has an impact on the hit rate. The more uniform the x- and y-dimensions in size, the better the hit rate. One explanation for this observation is that an uneven choice of dimensions increases the probability for over-fetching, i.e. data from memory is transferred to cache which is actually not needed.

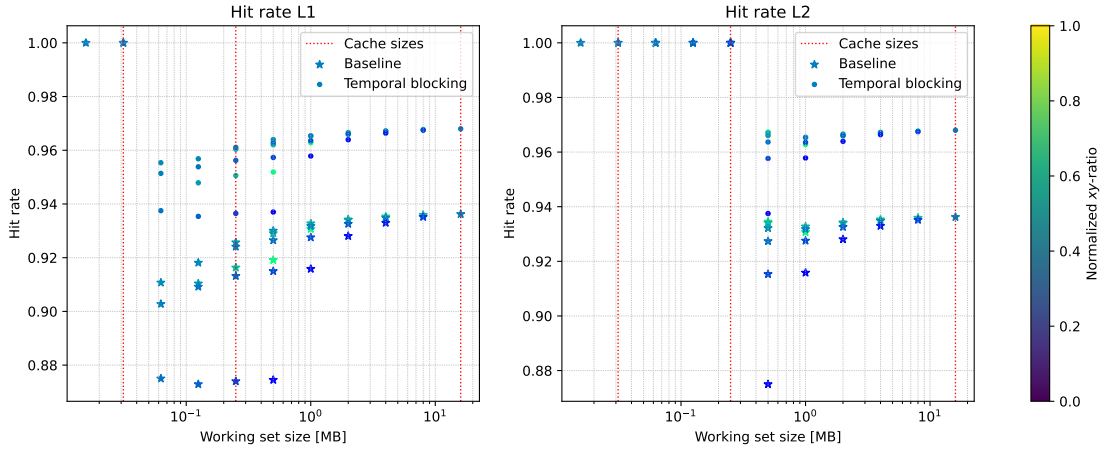

Figure 4: Recorded hit rates in the L1 (left) and L2 (right) cache for the 2D stencil. Star markers represent the baseline and circles the temporal blocking algorithm. The color indicates the equality of the two dimensions: a value of 0.5 means that x and y dimension are equal.

Conclusion

We have successfully implemented a cache simulator and shown that it can be used to justify the runtime of stencil codes. Additionally, we have used spatial and temporal blocking for various stencils and observed mixed results. For spatial blocking, the effective runtime seems to be rather sensitive to the choice of block sizes, as expected. Given our choice of stencils and the choice of the parameter space, barely any to none speedup has been observed. Conversely, temporal blocking shows more promising results with significantly higher hit rates. Time adds one more dimension which in our case is enough to leverage cache efficiency through temporal blocking.

This work could be extended in many ways. First and foremost, more stencils and a larger parameter space could be examined. Moreover, blocking could be extended towards the third spatial dimension, which may have interesting effects on higher-dimensional stencils. Spatial and temporal blocking could even be interleaved such that the time step is applied as soon as possible, which is equivalent to temporal blocking with increasing block size. Finally, the cache simulator could be expanded. A natural extension would be to track L3 cache accesses as well. Additionally, its current implementation is only an abstraction of loads and stores of memory addresses. This knowledge may be used to reason about other important effects such as vectorization or even ILP. Yet, caching together with locality has proven to be a main factor for performance on state-of-the-art machines, which is likely to stay this way in the near future.

Appendix

Reproducibility

This section deals with further details concerning the implementation and the methodology for the sake of reproducibility. We use Git for version control, and the source code is available upon

request. All numerical experiments are run on a single Intel i9-9900K CPU core with minimal background load, disabled Intel Turbo Boost Technology and enabled Intel Hyper-Threading Technology. Furthermore, all measurements are warm cache measurements averaged over 100 runs for the runtime and 10 runs for the hit rates. For the cache simulator, only the LRU policy is used, and we do not make use of the cold miss rate functionality. The correctness of all stencils is verified with the GoogleTest library directly in the code together with a Matlab implementation using convolutional filters.

We provide a list of used hardware and software components in the following:

Component	Value
CPU	Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Operating system	Ubuntu 23.04
Kernel version	6.2.0-32-generic
C++ standard	C++20
Compiler	g++ (Ubuntu 13.2.0-3ubuntu1 23.04) 13.2.0
Compiler flags	-DNDEBUG -O3 -ffast-math -fno-tree-vectorize
Eigen	Version 3.4.0
GoogleTest	Version 1.14
Matlab	Version R2022b

The parameter space consists of the number of grid points in each dimension, the number of halo points in each dimension, and the block size. For the spatial blocking experiments, the baseline and the blocking versions use the following parameters, where the blocking variables have no effect for the baseline:

```
nx_vals  = {4, 6, 8, 16, 32, 64, 128}
ny_vals  = {4, 6, 8, 16, 32, 64, 128}
nz_vals  = {64}
halo_nx  = {1}
halo_ny  = {1}
halo_nz  = {1}
block_nx = {16}
block_ny = {16}
```

For the blocking analysis, the following parameters are used:

For the temporal blocking experiments, the baseline and the blocking versions use the following parameters:

```
nx_vals  = {128}
ny_vals  = {128}
nz_vals  = {32}
halo_nx  = {1}
halo_ny  = {1}
halo_nz  = {1}
block_nx = {4, 8, 16, 32, 64, 128}
block_ny = {4, 8, 16, 32, 64, 128}
```

```
nx_vals    = {4, 8, 16, 32, 64, 128}
ny_vals    = {4, 8, 16, 32, 64, 128}
nz_vals    = {32}
halo_nx    = {1}
halo_ny    = {1}
halo_nz    = {1}
time_steps = {16}
```

Additional results

In the following, we provide additional results concerning spatial blocking.

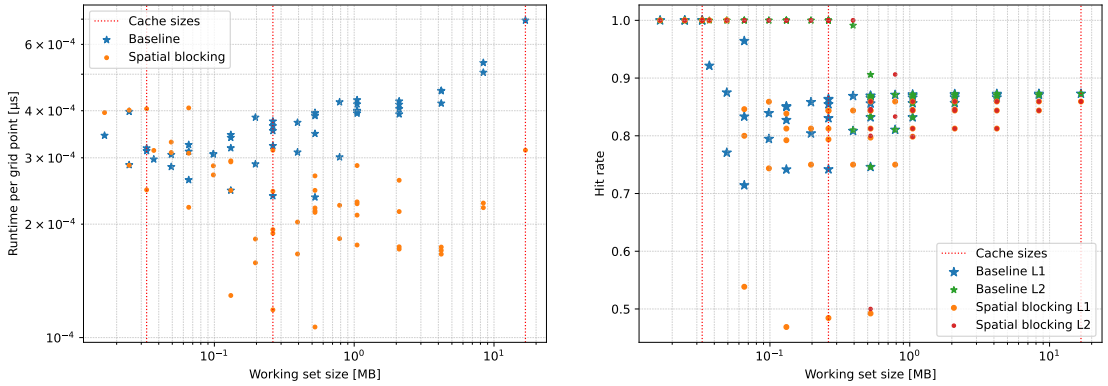

Figure 5: Stencil (1) runtime per grid point (left) and cache hit rates (right) for variable working set size.

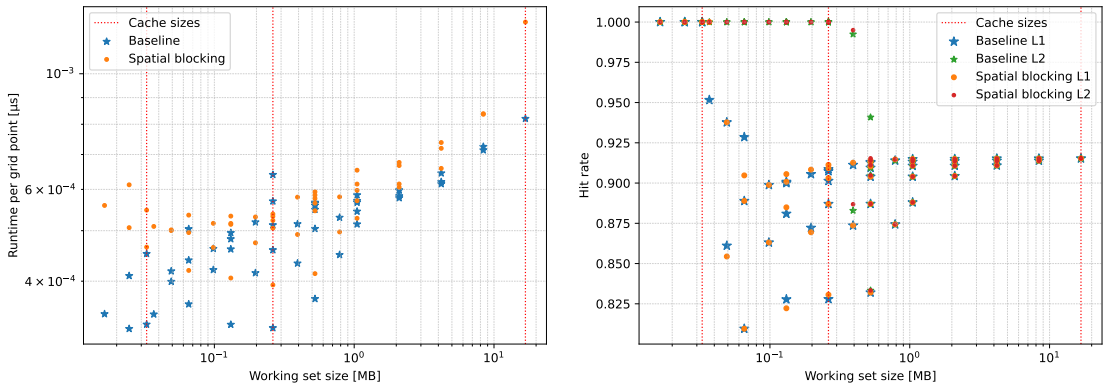

Figure 6: Stencil (2) runtime per grid point (left) and cache hit rates (right) for variable working set size.

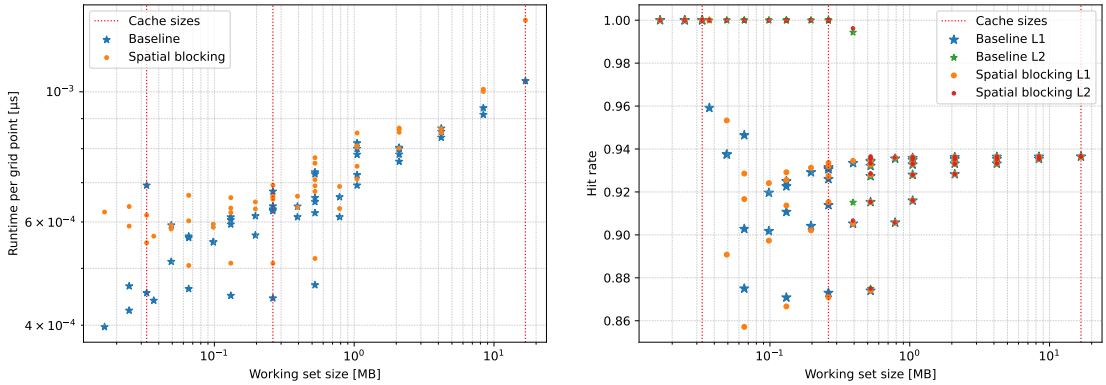

Figure 7: Stencil (3) runtime per grid point (left) and cache hit rates (right) for variable working set size.

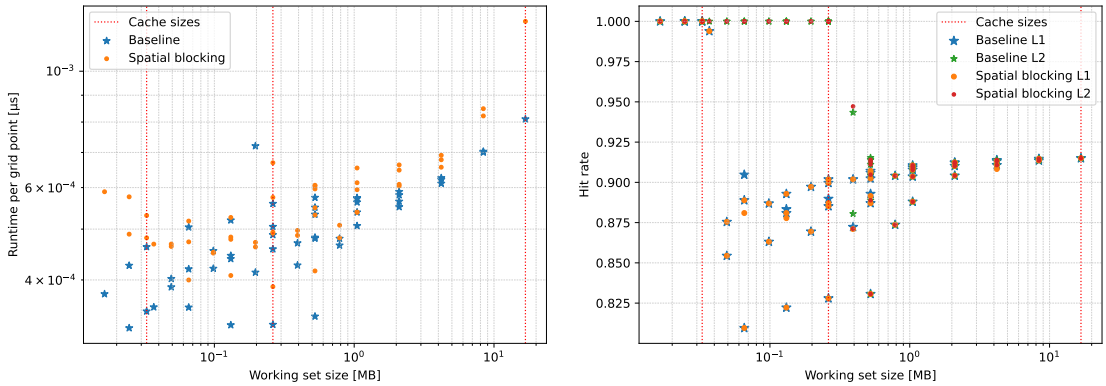

Figure 8: Stencil (4) runtime per grid point (left) and cache hit rates (right) for variable working set size.

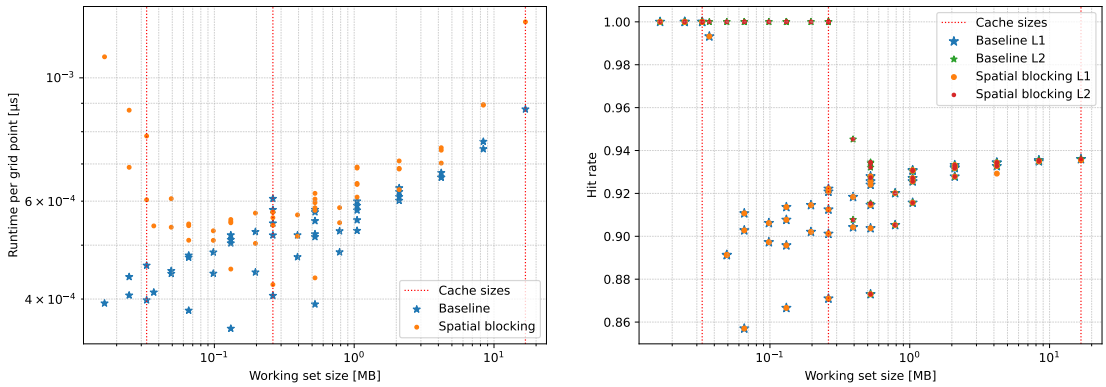

Figure 9: Stencil (5) runtime per grid point (left) and cache hit rates (right) for variable working set size.

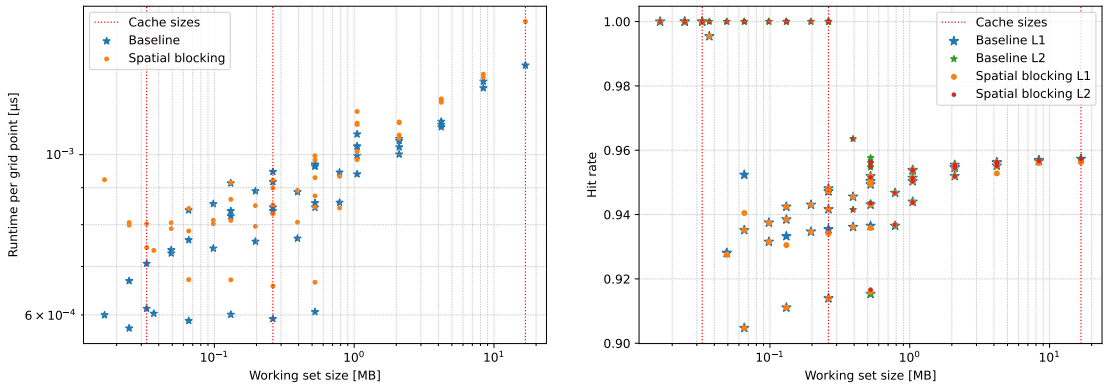

Figure 10: Stencil (6) runtime per grid point (left) and cache hit rates (right) for variable working set size.

References

[1] Joaquim Peiró and Spencer Sherwin, "Finite difference, finite element and finite volume methods for partial differential equations," in *Handbook of Materials Modeling*, pp. 2415–2446. Springer Netherlands, 2005.

[2] Bhishman Desai, Manish Paliwal, and Kapil Kumar Nagwanshi, "Study on image filtering – techniques, algorithm and applications," 2022.

[3] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas C. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, mar 2014.

[4] Michael Flynn, "Computer architecture," dec 2007.

[5] Randal E. Bryant and David R. O'Hallaron, "Computer systems," 2016.

[6] Gaël Guennebaud, Benoît Jacob, et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.