Cache-Aware Matrix Polynomials

Dominik Huber[1], Martin Schreiber[1][0000−0002−2390−6716], Dai Yang[2], and Martin Schulz[1]

[1] Technical University of Munich, Department of Informatics
domi.huber@tum.de, martin.schreiber@tum.de, schulzm@in.tum.de
[2] NVIDIA, Munich, Germany daiy@nvidia.com

Abstract. Efficient solvers for partial differential equations are among the most important areas of algorithmic research in high-performance computing. In this paper we present a new optimization for solving linear autonomous partial differential equations. Our approach is based on polynomial approximations for exponential time integration, which involves the computation of matrix polynomial terms ($M^p v$) in every time step. This operation is very memory intensive and requires targeted optimizations. In our approach, we exploit the cache-hierarchy of modern computer architectures using a temporal cache blocking approach over the matrix polynomial terms.
We develop two single-core implementations realizing cache blocking over several sparse matrix-vector multiplications of the polynomial approximation and compare it to a reference method that performs the computation in the traditional iterative way. We evaluate our approach on three different hardware platforms and for a wide range of different matrices and demonstrate that our approach achieves time savings of up to 50% for a large number of matrices. This is especially the case on platforms with large caches, significantly increasing the performance to solve linear autonomous differential equations.

Keywords: cache-blocking in time dimension · matrix exponentiation · higher-order time integration

1 Introduction

Solving time-depending partial differential equations (PDEs) on large-scale supercomputers is extremely resource demanding, yet applications demand the ability to operate on increasingly larger and more complex systems. Consequently, the development of efficient parallel PDE solvers from the mathematical side, as well as their efficient implementation on high-performance computing (HPC) systems is an active area of research. In this work, we investigate optimizations along the time dimension combining new approaches from mathematics and HPC research.

Our main application focus lies on linear autonomous PDEs that occur frequently, e.g., in full waveform inversion problems [9] or as part of splitting

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_10

methods that incorporate non-linear parts in a separate way [8]. In general, such PDEs are given by $\frac{\partial U(t)}{\partial t} = \mathcal{L}U(t)$ with $\mathcal{L}$ being the linear operator and $U(t)$ the solution at time t.

In order to solve such systems numerically for a given initial condition $U(0)$, we must apply a discretization. In particular, our presented HPC algorithms target commonly used discretization methods leading to a linear operator directly and explicitly given by a sparse matrix L. This is, e.g., the case when using discretizations based on finite differences or radial basis functions.

Furthermore, the discrete state of the solution at time t is given by $\boldsymbol{U}(t)$, leading to $\frac{\partial \boldsymbol{U}(t)}{\partial t} = L\boldsymbol{U}(t)$. Such a discretization typically results in sparse matrices that are then used in matrix-vector-like computations $L\boldsymbol{U}$, as it is common in off-the-shelf time integration methods. To provide an example, explicit Runge-Kutta (RK) methods rely on computations of the form $k_i = L\left(t_n + \Delta t c_i, \boldsymbol{U}^n + \Delta t \sum_j a_{i,j} k_j\right)$ with $\boldsymbol{U}^n$ being the approximated solution at time t_n, k_j related to the j-th RK stage, $a_{i,j}$ an entry in the Butcher table (e.g., see [1]) and Δt the time step size as part of the time discretization. However, such a formulation targets non-autonomous systems with the assumption of $\mathcal{L}(t)$ varying over time, e.g., by external time-varying forces, hence involving the dependency on time via $t_n + \Delta t c_i$.

In contrast, the linear PDEs we target in this paper do not involve any time-depending terms and this is indeed the case for many other PDEs (Seismic waves, Tsunami simulations, etc.). This opens up a new branch of matrix-polynomial-based time integration methods of the form $\boldsymbol{U}(t + \Delta t) = \sum_n \alpha_n (\Delta t L)^n \boldsymbol{U}(t)$, which we explore in this paper as the target for our algorithmic HPC optimizations. Similarly to the RK-based methods, these methods rely on matrix-vector products.

For their efficient implementation, though, we need to take modern HPC architectures into account, in particular their cache and memory hierarchy. We, therefore, design and implement a novel temporal cache-blocking scheme over the linear operators L as part of such a matrix polynomial computation. This increases both spatial and temporal locality and leads to a high utilization of the cache resources, leading to a speed-up of up to 50% on some architectures.

Our main contributions are the development of these caching strategies in Sec. 3, an analytical performance model which is presented in Sec. 4 as well as the performance assessment in Sec. 5.

2 Related Work

The growing gap between computational performance and memory access latencies and bandwidth, commonly referred to as the memory wall [12], is one of the fundamental bottlenecks in modern computer architectures. Caches are commonly used to mitigate this problem, but require careful algorithm design to achieve the needed temporal and spatial locality that makes their use efficient. This is particularly true for PDE solvers, which we target in this paper. Algorithm design and optimization is, therefore, an active and wide field of research. In the following we point out the most relevant related work and contrast it to our approach.

We first discuss very common optimization approaches for spatial dimensions. For matrix-vector and matrix-matrix multiplications, cache-blocking [6] is a well established technique and considered an essential optimization on today's architectures with deep memory hierarchies. For regular grid structures, this technique can be combined with tiling approaches, like spatial tiling [7], to further increase its efficiency. However, so far such optimizations only targeted the execution of a single operation, ignoring potential optimizations across multiple operators.

When considering the time dimension, temporal tiling and wavefront computations, as a generalization of it, has been shown to provide significantly improved performance on modern architectures [2, 13, 11]. In our work we build on this approach of temporal tiling, used for individual SpMvs, and apply it to a series of successive SpMvs, as they occur during the calculation of the matrix potentials $M^p\boldsymbol{v}$ needed for our class of targeted PDEs.

Contrary to stencil computations, our algorithms do not perform blocking over several time steps, but rather several sparse matrix-vector multiplications (SpMvs) computing the polynomial terms (vectors) in every time step. Furthermore, our approach can also be applied out-of-the-box to non-uniform grids. For temporal tiling, this would pose new requirements on data dependencies, as it is based on the explicit use of the regular grid structure.

Within the scope of the project to develop "communication-avoiding Krylov subspace methods" several publications focus on comparable approaches (see Hoemmen [4] and references therein). One particular difference of our work is the application of this technique in polynomial time integration. We also provide two different implementations of this technique, which enable the cache blocking naturally with a very small preprocessing overhead.

3 Cache-aware Matrix Polynomials

In this section we present two cache-aware algorithms for the calculation of the matrix polynomial terms $M^p\boldsymbol{v}$: the Forward Blocking Method (FBM) and the Backward Blocking Method (BBM). In particular, matrix-polynomial-based time integration demands not only the vector $M^p\boldsymbol{v}$ to be calculated, but rather all vectors $M^k\boldsymbol{v}$, $k \in \{1, \cdots, p\}$, which makes it infeasible to explicitly precompute the matrices M^k before multiplying them with $\boldsymbol{v}$. Therefore, these vectors are computed by typically successive matrix-vector multiplications with the same matrix M. With y_n denoting the result vector of the calculation of $M^n\boldsymbol{v}$, the vector $\boldsymbol{y}_{n+1}$ is derived as $\boldsymbol{y}_{n+1} = M^{n+1}\boldsymbol{v} = M\boldsymbol{y}_n$. This leads to the intuitive way to calculate $M^p\boldsymbol{v}$ by successive matrix-vector multiplications, i.e., the vectors $\boldsymbol{y}_1$ to $\boldsymbol{y}_p$ are calculated one after the other. We refer to this as the naive approach.

However, for sufficiently large problem sizes this results in no data reuse between the matrix-vector products, as the data is already evicted from the cache before it can be used again in the next multiplication. To avoid this situation our two methods use a blocking technique that enables the reuse of data over multiple matrix-vector calculations, which borrows some ideas from wavefront strategies in stencil computations. We interpret the vectors $\boldsymbol{y}_1$ to $\boldsymbol{y}_p$ as one-dimensional domains at time steps 1 to p, similar to one-dimensional stencil computations. While in such stencil computations the dependencies between the time steps are given by the defined stencil. for our calculations these dependencies are defined by the positions of nonzero entries in every row of the matrix. For matrices arising from finite differences or radial basis function discretizations, these positions are usually regionally similar in neighboring rows. Based on this observation, we apply a blocking scheme to the matrix to describe dependencies between whole blocks of the vectors. Our algorithms then construct two-dimensional space-time tiles over the vectors that fit into cache, while simultaneously respecting the dependencies between all blocks of the vectors.

To achieve this, our two methods use two different concepts: FBM starts at the first vector $\boldsymbol{y}_1$ and calculates successive blocks of a vector $\boldsymbol{y}_n$ until the dependencies for a block on the next vector $\boldsymbol{y}_{n+1}$ are fulfilled. BBM, on the other hand, starts at the last vector $\boldsymbol{y}_p$ and is stepping backwards through the dependency graph in a recursive way to calculate exactly the blocks needed to resolve the dependencies for the current block of $\boldsymbol{y}_p$. To realize these two concepts, both methods demand distinct information about the dependencies between the vector blocks. Therefore, we use different data structures for the FBM and BBM, as discussed next.

3.1 Extended CSR Matrix Formats

As a basis for our cache-aware matrix polynomial scheme, we extended the CSR matrix storage format to provide additional information about the non-zero block structure of the matrix. The CSR format uses three arrays: the non-zero entries of the matrix in the array `val`, the corresponding column indices in the array `colInd` and the row information as pointers into the two other arrays in the array `rowPtr`. We extended this format by (conceptually) partitioning the matrix into blocks of size $B \times B$, while keeping the underlying data layout of the CSR format. The information about the non-zero block structure is then stored in additional arrays. However, we use different formats for the two methods: while we store the positions of all non-zero blocks for the BBM, only the position of one non-zero block per *blockRow* has to be stored for FBM.

Therefore, for FBM the CSR format is extended by only one additional array of size $\lceil \frac{n}{B} \rceil$ for an $M^{n\times m}$ matrix. In this array the maximum *block-column* index of every *block-row* is stored (see `maxBlockColInd` array in Fig. 1). Hence, only a relatively small overhead of additional data has to be stored and loaded.

The format used by BBM, on the other hand, provides the full information about the non-zero block structure of the matrix. This information is stored in

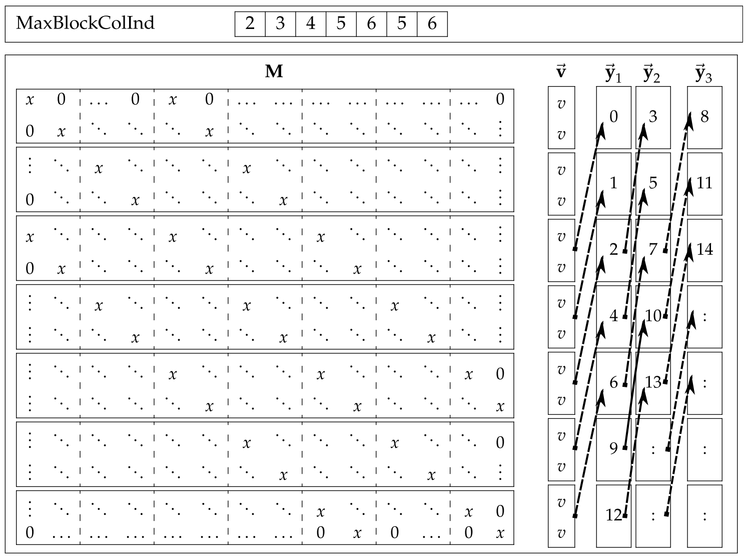


Fig. 1: **Forward Blocking Method**: This example shows the concept of the FBM for the calculation of $M^3 \boldsymbol{v}$ with sparse matrix M (left), the dense source vector $\boldsymbol{v}/\boldsymbol{y_0}$ (most left vector) and a block size of $B = 2$. The following vectors $\boldsymbol{y}_1$, $\boldsymbol{y}_2$ and $\boldsymbol{y}_3$ are the destination vectors of the successive SpMv operations needed to calculate $M^3\boldsymbol{v}$. The *numbers* inside the blocks of the destination vectors denote the order in which they are calculated by the FBM. The arrows indicate the dependencies between the vector blocks as encoded by the `MaxBlockColInd` array.

two arrays similarly to the `colInd` and `rowPtr` arrays of the CSR format, but by dealing with all block rows and columns instead of single ones (see Fig. 2). Thus, the `blockRowPtr` array consists of offsets into the `blockColumnIndex` array, indicating the start of a *block row*. The `blockColumnIndex` array contains the *block-column* indices of non-zero blocks in a *block-row*. If B_r denotes the number of non-zero $B \times B$ blocks of an $M^{n \times m}$ matrix, the size of the two arrays is given by $\lceil \frac{n}{B} \rceil$ and B_r.

3.2 The Forward Blocking Method

We implemented FBM according to the pseudo code in Alg. 1. For a better understanding of the underlying concept of this method, we present an example in Fig. 1. Based on this example we describe the algorithm while referring to the corresponding lines of code.

For each vector $\boldsymbol{y}_1$, $\boldsymbol{y}_2$ and $\boldsymbol{y}_3$ we track the information about its last calculated block (starting at -1) and the maximum index of the block of the predecessor vector that is needed to calculate the next block. For simplicity, in Alg. 1 we compute these values by the function calls `lastBlockOf(`$\boldsymbol{y}_n$`)` and `neededBlockFor(`$\boldsymbol{y}_n$`)`, respectively.

Algorithm 1 Forward Blocking Method: Calculates $y_p = M^p y_0$, where M is partitioned into *numBlocks* slices represented in the format described in Sec. 3.1

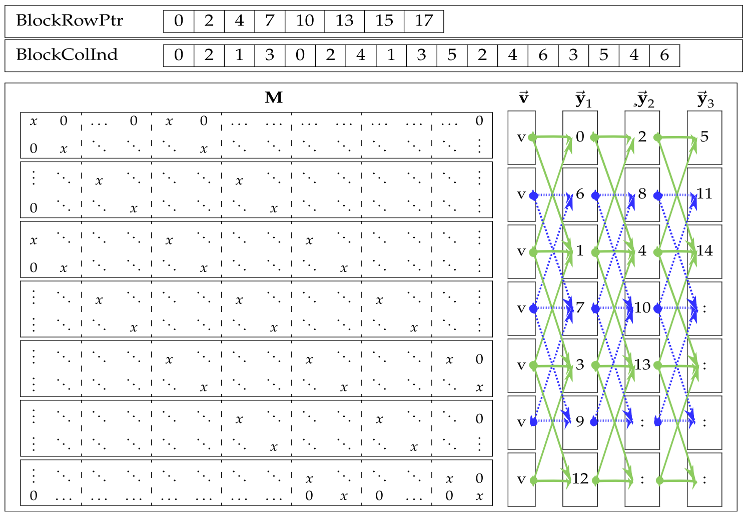
```
Require: $y_0$ is the source vector $\rightarrow$ lastBlockOf($y_0$) = numBlocks and $y_i, i \in [1, p]$ are
    empty vectors $\rightarrow$ lastBlockOf($y_i$)$= -1$, neededBlockFor($y_i$) $=$ maxBlockColInd[0]
 1: function FBM(numBlocks)
 2:     $p0 \leftarrow 1$
 3:     while neededBlockFor($y_p$) $! = -1$ do
 4:         for $i = p0$ to p do
 5:             while lastBlockOf( $y_{i-1}$ ) $\geq$ neededBlockFor($y_i$) and not ($i+1 <= p$ and
    lastBlockOf($y_i$) $>=$ neededBlockFor($y_{i+1}$)) do
 6:                 SpMv($y_i$, lastBlockOf($y_i$)+1)
 7:                 lastBlockOf($y_i$)++
 8:                 if lastBlockOf($y_i$) $<$ numBlocks $-1$ then
 9:                     neededBlockFor($y_i$) $=$ maxBlockColInd[lastBlockOf($y_i$)+1]
10:                 else
11:                     $p0++$
12:                     neededBlockFor($y_i$)=-1
13:                     break
```

Starting at $\boldsymbol{y}_1$, FBM loops through the vectors (Line 2 & 3), thereby calculating blocks of vector $\boldsymbol{y}_n$ by an arbitrary SpMv kernel (Line 5) until one of the following two conditions is reached (Line 4):

- The forward pointer in the last calculated block points to an unfilled block of $\boldsymbol{y}_{n+1}$: this indicates that the currently calculated data can be used to calculate a block of vector $\boldsymbol{y}_{n+1}$ and, therefore, the loop jumps to the next vector to propagate the new data forward.
- The forward pointer to the next block of $\boldsymbol{y}_n$ to be calculated originates from an unfilled block of y_{n-1}: this indicates that there are more blocks of the previous vector(s) needed, so the loop starts again at vector y_1.

When a block of vector $\boldsymbol{y}_n$ with index B_i is calculated, the value of `lastBlockOf`($\boldsymbol{y}_n$) has to be incremented (Line 7) and the new value of `neededBlockFor`($\boldsymbol{y}_n$) can be read from `maxBlockColInd`[$B_i + 1$] (Line 9). Completely filled vectors are excluded from the loop (Line 11). This loop is repeated until the last block of $\boldsymbol{y}_3$ is filled (Line 1).

The numbers in the vectors in Fig. 1 illustrate the order in which the blocks of the vectors would be calculated by FBM in this example. This order exhibits improved temporal locality on both the matrix and the vectors, compared to successive matrix-vector products, as it traverses the $dim \times p$-plane of the vectors in wavefronts with a certain (constant) wavefront angle, resembling those in stencil computation. It can be observed that the minimum tile size is dependent on the distances between the lowest and highest column index in every row. Hence, for very large distances FBM produces large space-time tiles to respect these dependencies, which impedes cache usage and, among other issues, excludes periodic boundary problems from the application domain of this method.



BlockRowPtr	0	2	4	7	10	13	15	17									
BlockColInd	0	2	1	3	0	2	4	1	3	5	2	4	6	3	5	4	6

Fig. 2: **Backward Blocking Method**: This example shows the concept of the BBM for the calculation of $M^3\boldsymbol{v}$ with sparse matrix M (left), the dense source vector $\boldsymbol{v}/\boldsymbol{y_0}$ (most left vector) and a block size of $B = 2$. The following vectors $\boldsymbol{y}_1$, $\boldsymbol{y}_2$ and $\boldsymbol{y}_3$ are the destination vectors of the successive SpMv operations needed to calculate $M^3\boldsymbol{v}$. The numbers inside the blocks of these destination vectors denote the order in which they are calculated by the BBM. The arrows indicate the dependencies between the vector blocks as encoded by the arrays `blockRowPtr` and `blockColInd`. The BBM computes a block of $\boldsymbol{y}_3$ by recursively computing all the (not already computed) blocks of the previous vectors it depends on, e.g., to calculate the first block of $\boldsymbol{y}_3$ ($\boldsymbol{y}_3[0]$) the order of calculated blocks is $\boldsymbol{y}_1[0]$, $\boldsymbol{y}_1[2]$, $\boldsymbol{y}_2[0]$, $\boldsymbol{y}_1[4]$, $\boldsymbol{y}_2[2]$, $\boldsymbol{y}_3[0]$.

Algorithm 2 Calculates block B_i of vector y_n recursively

```
1: function LOOKUPREC($y_n$, $B_i$)
2:     if $n > 1$ then
3:         for index=blockRowPtr[$B_i$] to blockRowPtr[$B_{i+1}$]$-1$ do
4:             $B_{rec}$ =blockColInd[index]
5:             if $\boldsymbol{y}_{n-1}$[$B_{rec}$].isEmpty() then
6:                 lookupRec($vecy_{n-1}$, $B_{rec}$)
7:     SPMV($y_n$, $B_i$)
```

3.3 Concept of Backward Blocking

Fig. 2 shows the concept of BBM.It loops over the blocks of the final result vector y_p ($\boldsymbol{y}_3$ in our example) and calculates the necessary blocks of the previous vectors recursively by calling the functions shown in Alg. 2. As input parameters, this function takes a vector $\boldsymbol{y}_n$ and the index B_i of the block of this vector that will be calculated. To calculate this block $\boldsymbol{y}_n[B_i]$, all the blocks of $\boldsymbol{y}_{n-1}$ from which pointers lead to $\boldsymbol{y}_n[B_i]$ are needed. The indices of these blocks can simply be read from the entries `blockColInd[blockRowPtr[`B_i`]]` to `blockColInd[blockRowPtr[`B_{i+1}`]-1]` (Line 3). If such a block of $\boldsymbol{y}_{n-1}$ is not calculated, yet, a recursive function call is performed for this block index and vector $\boldsymbol{y}_{n-1}$ (Line 5 & 6). When all necessary blocks are filled, $\boldsymbol{y}_n[B_i]$ can finally be calculated using a SpMv kernel (Line 10). The algorithm is effectively stepping backwards through the dependence graph in a depth first traversal to reach the needed filled blocks and is calculating exactly the required data on the way backtracking forward through the dependence graph.

As above, the correct order of the calculations of the vector blocks improves the temporal locality of the data accesses. For the shown example of a regular grid, BBM calculates blocks of vectors (after a short initialization phase) in the same order as FBM. However, the additional information of the dependencies between the blocks leads to a decisive advantage. As discussed above FBM degenerates to nearly successive SpMv calculations for large distances between non-zeros in one or multiple rows. BBM can "compensate" a small number of such rows, if for a majority of rows these distances are small enough for the space-time tiles to fit into cache. For such cases, BBM breaks the wavefront analogy and only calculates exactly the needed data blocks. This is contrary to FBM, which calculates all blocks of a vector up to the maximum needed block.

4 Analytical Best-Case Performance Model

In order to understand the quality of our proposed solution, we derive an analytical model showing the upper bound for the performance improvements possible with our blocking methods. When calculating $M^p\boldsymbol{v}$ for large problem sizes without cache blocking, the values of the matrix and vectors have to be loaded from memory for every matrix-vector multiplication. Thus, the time for the naive calculation is given by $T_{naive}(p) = p \times T_{mem}$, where T_{mem} denotes the time needed for an SpMv with no values cached.

Our approaches use cache blocking, hence—in the ideal case—the matrix and vector values are only loaded once from memory and then reside in cache for the rest of the computation. Following this observation, we model the computation time as $T_{blocked}(p) = T_{mem} + (p-1) \times T_{cache}$, where T_{cache} is the time needed for in-cache SpMvs. The time savings through blocking can then be calculated as $T_{save}(p) = 1 - \frac{T_{blocked}(p)}{T_{naive}(p)}$. Consequently, for increasing exponents of the matrix (p), the expected time savings through blocking converge to $\lim_{p\to\infty} 1 - \frac{T_{mem}+(p-1)\times T_{cache}}{p\times T_{mem}} \approx 1 - \frac{T_{cache}}{T_{mem}}$.

The size of the data that has to fit into the cache to achieve full cache blocking (S_C) is heavily dependent on the specific matrix structure and the exponent of the matrix. I.e., the relation can be described as $S_C \propto R_{nz} \times B_w \times p$, where R_{nz} is the number of nonzero values per row, B_w the distance between the lowest and highest column index per row and p the exponent of the matrix. For regular grid based matrices, the computation order in which the two methods compute the blocks lead to a more accurate approximation of $S_C \approx \frac{B_w}{2}(p(R_{nz}(S_{val} + S_{ind}) + 3S_{val} + S_{ind}) + S_{val})$, where S_{val} is the size of the data type of the matrix/vector values and S_{ind} the size of the data type used for the index and pointer array of the CSR format.

5 Evaluation

5.1 Targeted Hardware Architectures

To analyze the effectiveness of our approach, we evaluate our approaches on three different hardware platforms: `XeonBronze, XeonSilver and AMD`. Both `XeonBronze` and `XeonSilver` are based on the Intel Skylake Architecture; the `XeonBronze` platform features an Intel Xeon Bronze 3106 8-core processor with a total of 80GiB of DRAM, and the `XeonSilver` platform is equipped with two Intel Xeon Silver 4116 12-core processors and a total of 96GiB of DRAM, arranged equally across all available memory slots to allow for optimal bandwidth. Our `AMD` platform is built with a single AMD Ryzen Threadripper 2990WX 32-core processor. It is based on the 2nd-generation AMD Zen architecture and features a total of 64 GiB of main memory.

The Intel Skylake [3] features a classical 3-layer cache design (L1I/D, L2 and L3), with each of the layers (L1I/D, L2 and L3) being non-inclusive. L1D and L1I caches are 32KiB large and 8-way associative, the L2 cache has a size of 1MiB and is 4-way associative, and all three caches are exclusive to a particular core. The L3, on the other hand, is a shared cache and has a size of 1.375MiB per core on our reference systems, resulting in a total of 11MiB (Xeon Bronze) and 16.5MiB (Xeon Silver) L3 cache shared betweeen the cores of a processor.

On AMD's 2nd-generation Zen architecture (Zen+) the L1I caches are 64KiB and the L1D are 32KiB per core and each core also has its own 256KiB L2 cache. Unlike on Skylake, the L1 caches are full inclusive with respect to the L2 caches. A special design of Zen+ is the so-called CCX consisting of a cluster of 4 cores, which each shares an 8MiB L3 cache. Two CCXs are located on one die and our reference platform (2990WX) features a total of 4 dies in its package. The dies are interconnected with a high-speed interconnect named *Infinity Fabric*. On the 2990WX, two memory controllers are attached to two of the dies, resulting in 4 NUMA domains, in which two of the domains do not have direct memory access and need to route accesses through another core.

5.2 Matrix Test Suite

The structure of the generated matrices depends on the particular grid, the finite difference order and the boundary condition. To identify the interplay between these parameters and our developed algorithms, we construct two matrix test suites that cover a wide range of combinations of these parameters. We give an overview of these matrices we used for our tests in Tables 1 and 2.

Test Suite 1 (TS 1)		
PDE	$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \right) = \alpha \nabla^2 \boldsymbol{u}$	
FD Orders	2, 4, 6, 8	
Boundary Condition	Homogenous Dirichlet	
	TS 1a	**TS 1b**
Matrix IDs	0 - 27	28 - 55
Dimensionality	2D	3D
Grid dimensions	224, 316, 447, 632, 775, 894, 1000	37, 46, 58, 74, 84, 93, 100

Table 1: Overview of matrices in Test Suite 1

Test Suite 2 (TS 2)			
Matrix IDs	56-136		
PDE	$U(\boldsymbol{x}, t) = c\nabla U(\boldsymbol{x}, t)\,, c = 1$		
FD Orders	2, 4, 6, 8		
Boundary Condition	Periodic, $U(\boldsymbol{x}, 0) = sin(2\pi(\boldsymbol{x} - \boldsymbol{x}_0))$		
Dimensionality	1D	2D	3D
Grid Dimensions	$2^n\,, n \in [5, 10]$	$2^n \times 2^{n-1}\,, n \in [6, 10]$	$2^n \times 2^{n-1} \times 2^{n-2}\,, n \in [7, 10]$

Table 2: Overview of matrices in Test Suite 2

5.3 Benchmark Description and Configuration

We investigate the behavior of our two methods for matrices of TS 1a and 1b. As FBM is not suited for problems with periodic boundary conditions (see Sec. 3.2), we test only BBM for matrices of TS 2.

We run tests using SSE4.2, AVX, AVX2 and AVX512 on the Intel systems and an AVX2 implementation on the AMD system, using block sizes of $B = 2^i, i \in \{6, \cdots, 12\}$. We further use an affinity of the single-threaded program to the core closest to the memory controller on each architecture. Our findings show that the differences in the vector instruction sets and the underlying micro architecture realizing them have a great impact on the performance of SpMvs with the matrices of our test suites: using AVX512 consistently leads to lower performance. Further, the performance of SSE, AVX and AVX2 instructions seem to be highly dependent on the specific matrix, making it difficult to get to a general conclusion on which vector extensions to use. Hence, if not further specified, we use the results of the best performing vector extension for our implementation and the reference method, respectively. We compare the results of our approaches to an implementation of the naive approach, which performs the matrix-vector multiplications sequentially (see Sec. 3). For this, we use the best performing block size evaluated per matrix and exponent. For all occurrences of SpMv calculations, we use the routine `mkl_sparse_d_mv()` of the Intel Math Kernel Library (MKL) [5].

We also use the same library on the AMD processor, although it often is reported to not reach high performance on non-Intel CPU types. Several factors led to this decision: by setting the environment variable `MKL_DEBUG_CPU_TYPE=5`, the library can be forced to choose the AVX2 code path instead of the default

SSE path to which it normally falls back to on non-Intel CPUs. Comparing the performance of the AVX2 code path to other libraries on our AMD architecture, e.g., OSKI [10], we found that for our cases the optimal library is dependent on the specific problem type and size. Moreover, this paper focuses on exploring the general potential of cache-aware algorithms for this type of calculation, rather than achieving overall maximum performance in using SpMvs directly, motivated further by the results in the following section. We therefore stick with MKL on all architectures.

5.4 Results

In this section we present our results of FBM and BBM introduced in Sec. 3. We measure the time needed for the calculation of $M^p \boldsymbol{v}$, $p \in \{2, 3, 4, 5\}$ and compare it to the reference implementation without cache blocking. Our results show improved performance of our blocking methods on all three architectures for a large number of matrices in the test suites.

For the matrices of TS 1a and 1b, both FBM and BBM produce quite similar performance behavior; consequently, these results are shown interchangeably in Fig. 3. For most of the matrices in TS 1a, our approaches outperform the reference method. We achieve time savings of up to 25% / 15% on the Intel Xeon Bronze/Silver, respectively, and up to 50% on the AMD Ryzen Threadripper. The matrices in TS 1b lead to slightly less improvements on the Intel processors, but still produced time savings of up to ±15% and ±5%, respectively. On the AMD, we measure greater performance improvements of 40% to 50% for many of these matrices.

Regarding the periodic boundary problems of TS 2, BBM still achieves the same kind of performance improvements as for some matrices resulting from 2D FD grids (Figs. 4).

5.5 Evaluation Compared to the Analytical Model

On all three hardware platforms, we measure the in-L2/L3-cache and in-memory performance for SpMvs with matrices similar to those in the test suites and then use these values in our analytical model as described in Sec. 4. The upper bounds for the time savings of our blocking approach derived from the model are 20%/12% on Intel Xeon Silver, 30%/15% on Intel Xeon Bronze and 55%/50% on the AMD Ryzen Threadripper, which closely resembles our real measured performance.

The performance of our approaches depends on size and structure of the matrix as discussed in Sec. 4. Using cache blocking, these algorithms naturally can only provide significant performance improvements if the matrix itself does not fit into cache. Moreover, the matrix properties B_w and R_{nz} have to be small enough such that the space-time tiles do fit into cache. This explains the poor performance of the methods for very small matrices (e.g., matrices 0, 1 and 2) and matrices with large B_w and R_{nz} (e.g., matrices 53, 54 and 55), while they perform well for large sparse matrices.

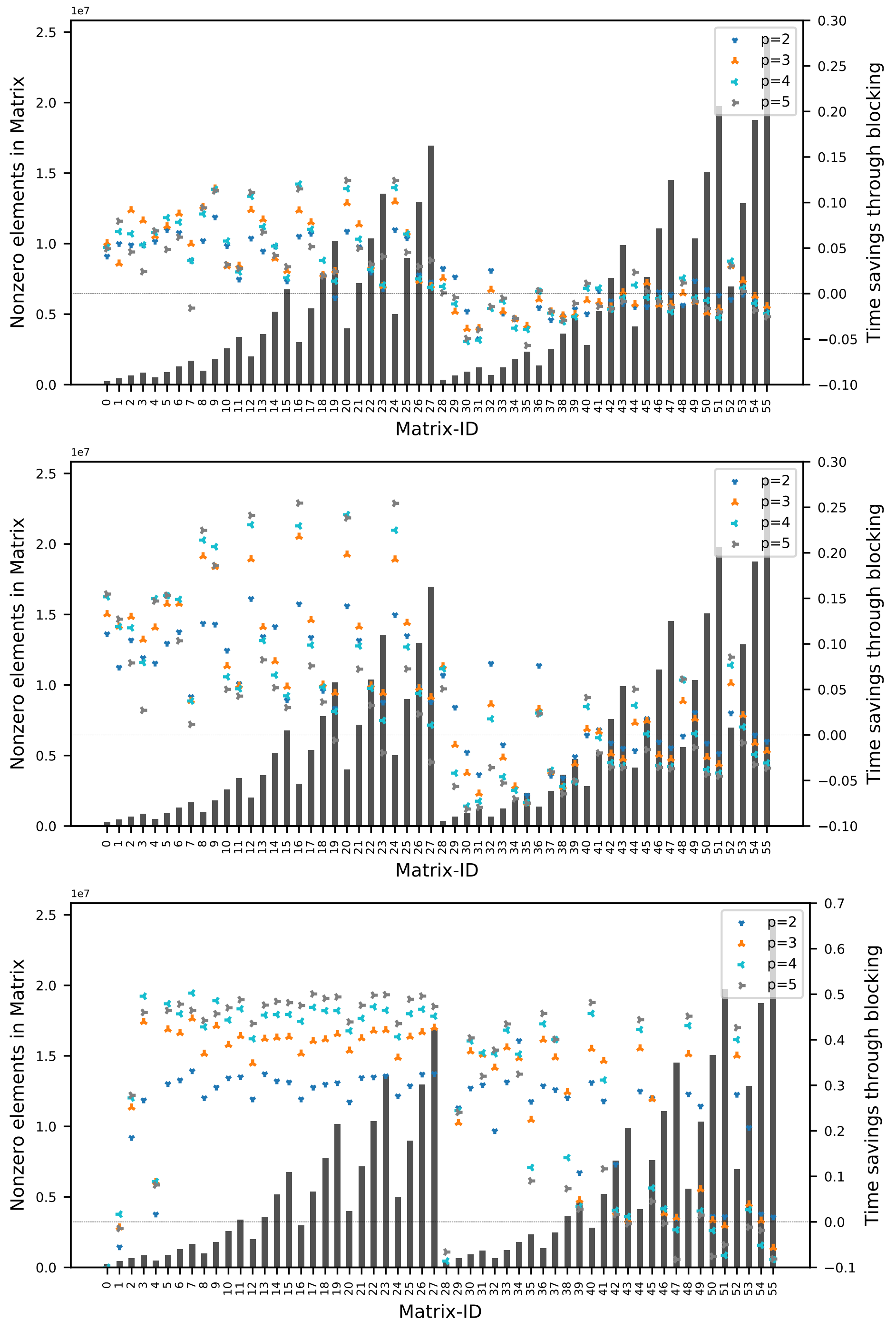


Fig. 3: Performance improvement of BBM/FBM on Xeon Silver (top), Xeon Bronze (mid) and AMD Ryzen Threadripper (bottom) for matrices in TS 1a and 1b (Dirichlet boundary condition): time reduction through blocking using best performing ISA for both, BBM/FBM and the reference method. The number of floating-point operations is $2 \times p \times$ *number of non-zeros*.

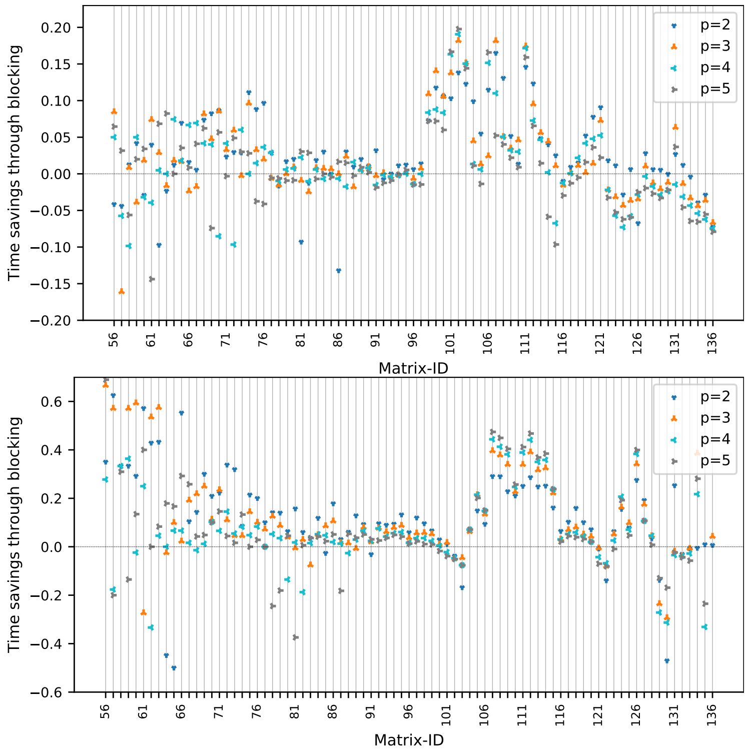


Fig. 4: Performance improvement of BBM on Intel Xeon Bronze (top) and AMD Ryzen Threadripper (bottom) for TS 2 (periodic boundary condition): time savings through blocking using AVX2 for both, the BBM and reference method.

6 Summary and Discussion

In this paper we investigated the potential of using cache aware algorithms to increase the performance of matrix polynomials in the context of higher-order time integration of linear autonomous PDEs. We introduced two algorithms: the Forward Blocking Method (FBM) and the Backward Blocking Method (BBM), both using a cache blocking technique to allow data reuse during the calculation of $M^p v$.

Our evaluation on three different architectures showed both methods profit from larger and faster caches. Further, our approaches showed improved performance for a large number of matrices of our test suites. These are matrices not fitting into cache, while the generated space-time tiles do. We showed that the ratio of in-cache and in-memory SpMv is a good indicator for upper bounds of the performance improvements of our method to be expected on a specific architecture. This is also the deciding factor, why better results can be observed especially on AMD by blocking for the L3 cache.

Our experiments showed that BBM is the more flexible approach. While FBM is (by design) not suited for periodic boundary problems, our results showed BBM also achieved improved performance for such matrices.

Overall, our results showed promising time savings of both methods compared to the standard approach of successive sparse matrix-vector multiplications. Therefore, our approach is a significant step towards further reducing the wallclock time of higher-order time integrators for linear autonomous partial differential equations.

Future work will extend these algorithms to exploit multi-core architectures. Here, various new challenges will arise, such as possible data races, which ultimately show up for such unstructured problems. However, also opportunities such as the exploitation of additional caches can lead to further performance boosts. Additionally, future work will leverage the performance boosts of the presented algorithms in the context of time integrating PDEs.

References

1. Butcher, J.C.: Implicit Runge-Kutta Processes **18**(86) (1964)
2. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review **51**(1), 129–159 (2009), http://www.jstor.org/stable/20454196
3. Doweck, J., Kao, W., Lu, A.K., Mandelblat, J., Rahatekar, A., Rappoport, L., Rotem, E., Yasin, A., Yoaz, A.: Inside 6th-generation intel core: New microarchitecture codenamed skylake. IEEE Micro **37**(2), 52–62 (Mar 2017)
4. Hoemmen, M.F.: Communication-avoiding Krylov subspace methods. Ph.D. thesis, EECS Department, University of California, Berkeley (Apr 2010), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html
5. Intel: Intel Math Kernel Library Developer Reference, revision: 023 edn. (9 2019)
6. Nishtala, R., Vuduc, R., Demmel, J., Yelick, K.: When cache blocking sparse matrix vector multiply works and why. Applicable Algebra in Engineering Communication and Computing **18**, 297–311 (01 2007)
7. Rivera, G., Tseng, C.W.: Tiling optimizations for 3d scientific computations. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing. p. 32es. SC 00, IEEE Computer Society, USA (2000)
8. Ruprecht, D., Speck, R.: Spectral Deferred Corrections with Fast-wave Slow-wave Splitting. SIAM Journal on Scientific Computing **38**(4), A2535–A2557 (2016)
9. Virieux, J., Asnaashari, A., Brossier, R., Métivier, L., Ribodetti, A., Zhou, W.: 6. An introduction to full waveform inversion (2014)
10. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series **16**, 521–530 (jan 2005)
11. Wellein, G., Hager, G., Zeiser, T., Wittmann, M., Fehske, H.: Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. vol. 1, pp. 579–586 (01 2009)
12. Wulf, W., McKee, S.A.: Hitting the memory wall: Implications of the obvious. Tech. rep., USA (1994)
13. Yount, C., Duran, A., Tobin, J.: Multi-level spatial and temporal tiling for efficient hpc stencil computation on many-core processors with large shared caches. Future Generation Computer Systems **92**, 903 – 919 (2019)