LOW LEVEL PERFORMANCE OPTIMIZATIONS AND VECTORIZATION OF THE SCALING AND SQUARING ALGORITHM FOR THE MATRIX EXPONENTIAL

Andrea Keusch, Krishna Le Moing, Philipp Engljähringer, Roman Marquart

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The primary focus of this paper revolves around the analysis of the matrix exponential function. To accomplish this, we employ the new and improved algorithm presented in the paper of `Higham` and `Al-Mohy`[1]. This algorithm uses the scaling and squaring method for computing the matrix exponential. Subsequently, we delve into a comprehensive optimization process to improve the performance of this function, aiming to identify potential bottlenecks and optimization opportunities. Moreover, we conduct an extensive testing phase to assess the overall impact of our optimizations on performance. We also compared our optimized version to a version which uses the `OpenBLAS` library for its matrix matrix multiplications. By undertaking these steps, we aim to gain insights into the efficiency and effectiveness of our optimizations.

1. INTRODUCTION

The matrix exponential is a power series defined as follows

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + ...$$

This function has a wide array of use cases, mainly in the domain of perturbation analysis, control, and parameter estimation of linear dynamical systems [2]. The versatility of the matrix exponential fostered a favorable environment for research. This research lead to the development of numerous methods for diverse use cases. In our paper we are using the scaling and squaring method. It is the most popular method for computing the matrix exponential, and can be found in Mathematica for example [1].

Contribution. In our paper we will commence by replicating the algorithm presented in the paper of `Awad H. Al-Mohy` and `Nicholas J. Higham` namely `A New Scaling and Squaring Algorithm for the Matrix Exponential` [1]. By establishing a solid foundation through our initial implementation, we will then embark on an extensive optimization phase. Our optimization strategies encompass a range of techniques, including : vectorization leveraging SIMD instructions, loop unrolling to enhance computational efficiency, pre-computation techniques to minimize redundant calculations, and algorithmic refinements aimed at improving overall performance.

Related work. As stated above, our work is based on the paper `A New Scaling and Squaring Algorithm for the Matrix Exponential` [1]. This paper focuses on the scaling and squaring method and is based on the approximation

$$e^A \approx (R_m(2^{-s}A))^{2^s}$$

where R_m is the $[m/m]$ Padé approximant to e^x. This new approach to the scaling and squaring method mainly tackles the case of overscaling by introducing two key ideas. In the context of triangular matrices, the initial improvement involves calculating diagonal elements during the squaring phase as exponentials rather than relying on powers of R_m. The second key idea is to modify the backward error analysis approach by utilizing the sequence $||A_k||^{1/k}$ instead of $||A||$ in algorithms. This is because, in the case of non-normal matrices, $||A_k||^{1/k}$ can be significantly smaller than $||A||$, particularly when overscaling occurs in current algorithms.

Another reference for our work is the paper `A block algorithm for Matrix 1-Norm estimation, with an application to 1-norm pseudospectra` from `Nicholas J. Higham` and `Françoise Tisseur` [3]. We use the algorithm 2.4 of this paper to calculate an estimate of the 1-norm of a matrix. The approach it proposes divides the matrix into blocks and samples columns from each block. It then computes the 1-norm of the sampled columns and combines them to obtain an overall estimate of the matrix's 1-norm.

2. THE MATRIX EXPONENTIAL ALGORITHM

In this section we provide a high-level overview of the algorithm. We omit details that are not relevant for the discussions of our optimizations later on. The full details can be found in [1], which also provides a pseudo-code in Algorithm 5.1.

The coarse structure of the matrix exponential algorithm under consideration is as follows. First we try to find the smallest degree $m \in \{3, 5, 7, 9\}$ such that the $[m/m]$ padé-approximant provides an accurate result. Choosing the smallest m is desirable because it is less expensive. The decision depends on some norms of the input matrix A, which are computed using algorithm 2.4 of Higham and Tisseur [3]. For matrices with small norms a lower padé-approximant is sufficient, while large norms require a higher approximation degree. If a suitable $m \in \{3, 5, 7, 9\}$ has been found, the padé-approximant can be computed directly. If not, the scaling and squaring technique has to be applied. We now describe both cases in more detail.

Computing the padé-approximant directly. In this case we have found a degree $m \in \{3, 5, 7, 9\}$ that provides enough accuracy. To compute the $[m/m]$ padé-approximant R_m, we first compute P_m and Q_m by scaling and adding up powers of A up to the degree m. The computation has the form $b_0 * I + b_1 * A + ... + b_m * A^m$, where b_i are the padé-coefficients. The exact definition can be found in [1], equation (3.4). The padé-approximant R_m can be derived by solving the linear system of equations (LSE) $Q_m * R_m = P_m$ (equation (3.6) in [1]).

Scaling and Squaring technique. In this case, choosing a degree $m \leq 9$ would lead to an inaccurate result. Therefore, we choose $m = 13$ which has been found to be the optimal degree and apply the scaling and squaring technique. First, the input matrix A is scaled down by multiplying all elements by a power-of-2 fractional (e.g. $2^{-s}A$ for some $s > 0$). Then we can compute P_m and Q_m using equation (3.5) in [1], which is similar to (3.4) and just sums up some scaled powers of A. The padé-approximant r_m can be computed by solving the LSE $Q_m * R_m = P_m$, as in the previous case. In the final step, the scaling has to be reversed. Due to the properties of R_m this results in the computation of $R_m^{2^s}$. As the exponent is a power of 2, repeatedly squaring A leads to the correct results with low cost. For triangular matrices this computation can be optimized further but we omit the details here and refer to code fragment 2.1 in [1] instead.

Cost Analysis. Analysing the cost of this matrix exponential algorithm is hard because it strongly depends on the input matrix A. Namely, we have the cases $m \in \{3, 5, 7, 9\}$, each with a different cost. Then there is the scaling and squaring technique with $m = 13$ which is more expensive. Additionally, for some computations triangular matrices allow for further optimizations. In order to get an accurate cost measurement, we implemented some macros that allow us to determine the cost for each input matrix individually.
When computing the cost, we only consider floating point operations and we do not distinguish between different operations. More concretely, additions, multiplications, divisions, comparisons and so on are all considered to have the same cost.
The asymptotic complexity is in $O(n^3)$ with a large constant, as we do a lot of matrix-matrix multiplications. Additionally, we need to solve a linear system of equations, which is also $O(n^3)$.

3. PERFORMANCE OPTIMIZATIONS OF THE SCALING AND SQUARING ALGORITHM

We now discuss the straightforward implementation of the matrix exponential algorithm and the optimizations we applied to boost the performance.
We split this section into different subsections on some of our optimizations of sub-algorithms of which our main algorithm consists.
One thing to mention at the beginning of this section is that our optimized version uses Jinja [4] templates to simplify the generation of different versions of our algorithm and loop unrolling. The generated templates are concatenated into one big file with all parts of the algorithm. Additionally, we assume that matrix sizes are multiples of 4 in the optimized version such that we can apply unrolling and vectorization without introducing remainder loops.
In both the baseline and the optimized case, we checked for correctness by comparing our results to the Matrix exponential implementation in the well known `Eigen` C++ library.

3.1. Matrix Exponential (main algorithm)

Base implementation. Our base implementation of the matrix exponential algorithm is very close to the pseudo code in the original paper. However we did make some changes. We pre-calculated some results which we would use later. Specifically the potentiations A^2, A^4 and A^6 as well as $|A|^2$, $|A|^4$ and $|A|^6$, which we then could use for the computation of (3.4) and (3.5).
Another algorithmic optimization is to check if the matrix is triangular at the start of the algorithm. The base implementation operates on row-wise stored matrices.

Optimizations. As most of our optimizations were made on sub-algorithms, we kept the main part of our algorithm relatively close to the base implementation. One optimization we made was inlining matrix exponentiation by squaring at the end of the algorithm and in the ℓ function. This was done to avoid copying the matrices in the function but the impact of this optimization is probably miniscule compared to the rest of the optimizations so we did not specifically analyze this.

The optimized version operates on column-wise stored matrices as this is favorable for the heavily used computations of 1-norms, see sections 3.3 and 3.4.

3.2. Matrix Matrix Multiplication

Matrix-matrix multiplications are a big part of our algorithm. While we implemented our own version, we were also able to use OpenBlas to speed up and compare our optimizations to it.

Base implementation. Our base implementation of the matrix matrix multiplication (mmm) is a simple triple loop operating on row-wise stored matrices. This leads to matrix A (the left hand side matrix) having a better access pattern. We also used a flag with which we could set if we use OpenBlas or not.

Optimizations. Our optimized version of mmm is vectorized and operates on 4 by 4 chunks of the matrix stored in column-major instead of just one element at a time. This choice was made because with vectorization, 4 doubles can be loaded at once. Because this implementation now operates on column major matrices, matrix B (the right hand side matrix) has the better access pattern. Again, we use a flag for our template which if set to true generates a version where we use OpenBlas instead of our implementation.

3.3. Matrix One Norm $||A||_1$

The matrix one norm is defined as $||A||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$. It finds the maximum of the sum of the absolute entries of all columns in the Matrix A.

Base implementation. The base implementation simply sums up the absolute values of all columns and returns the maximum of these sums. Because it operates on row-major order stored matrices and needs columns, cache locality is bad.

Optimizations. The change to column-major stored matrices alone is already an optimization for this algorithm, as it results in a linear access pattern and better cache locality. Additionally, we vectorized the algorithm. In each column, a vector is loaded, we find the absolute values of the elements of the vector and add it to the sum of the previous vectors in the column. In the end, each vector of sums is added together and the maximum is found. By unrolling these operations, multiple columns can be added together in parallel (using instruction level parallelism), see figure 1.

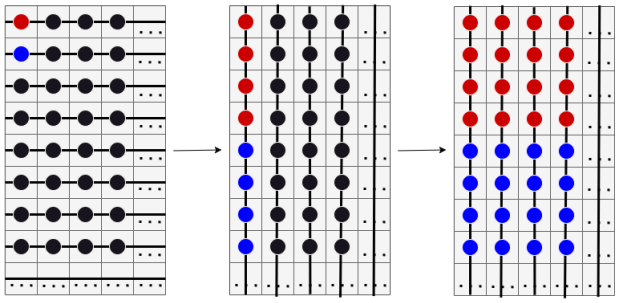

Fig. 1. Access pattern and vector opimizations of $||A||_1$

3.4. One Norm Estimation

The one norm estimation algorithm is defined by Higham and Tisseur in algorithm 2.4 [3]. It estimates the one norm of A such that $normest(A) \leq ||A||_1$

Base implementation. In the base implementation, we already changed the algorithm to make analysis a bit simpler. For example, the parameter t was set to 2. This choice allowed us to avoid calling an external sort function to find candidates for the indices of the columns with the norm and instead just pass through h linearly, see Algorithm 2.4 in [3], labels (3) and (4). Because the base implementation operates on row-wise stored matrices, the parallel columns check on S, see Algorithm 2.4 in [3], label (2), had to be done on transposed matrices. This introduced some overhead. There are also some calls to rand() in this part of the code. We did not include these in our flop count.

Optimizations. The optimized version contains a special vectorized matrix matrix multiplication algorithm for (n x n) * (n x 2) matrices. It uses the same scheme as the previously described mmm algorithm. Alternatively, we can set a flag and use OpenBlas instead. Due to the column-major format, we do not need to transpose the matrix anymore. Additionally, we vectorized the algorithm. We did not take too much time to optimize these functions as they operate on n x 2 matrices and therefore should not contribute to the overall performance of our main algorithm too much.

3.5. Computing (3.4) and (3.5)

The equations (3.4) and (3.5) described by Al-Mohy and Higham [1] compute P and Q, which are used in the system of linear equations that is later solved to get the padé-approximant of e^A. The two equations are similar. Some powers of the input matrix A are scaled by the padé-coefficients and then summed up. Depending on the padé-approximation powers up to 3, 5, 7, 9 or 13 are relevant. The only difference between (3.4) and (3.5) is that (3.5) is used only for m=13 and the computation is optimized such that as

few matrix-matrix multiplications as possible are required. Namely, only even powers of A are computed explicitly, see [1] for more details.

Base implementation. For the baseline, we just scale and add one matrix after another. All even powers of A are already pre-computed as they are also needed for some other computations. We use the schemes proposed by [1] to get higher powers of A and also for the uneven powers. Namely, we multiply intermediate matrices by A or A^6.

Scalar optimizations. First, we divided the method for (3.4) into 4 methods, each of them handling a special case, e.g. $m = 3, 5, 7, 9$. This removes some unnecessary branches but the main reason for this change was to simplify further optimizations. We focused on optimizing $m = 9, 13$ as these are the worst case scenarios.
The main idea for scalar optimizations is to optimize the cache access pattern. The powers of A are used multiple times. For example, in the case $m = 9$ A^2 is used to get $\beta_2 A^2$ and $\beta_3 A^3$. To reduce the number of memory loads we compute the intermediate results U and V simultaneously such that we iterate over each input matrix only once. Additionally, we can reduce the number of writes by fully fusing the computation of U and V into one single loop. The trade-off is that there could be conflict misses as we have quite a few matrices that are being accessed within one loop. For $m = 9$ there are 4 input matrices and 2 result matrices, all of size $n \times n$.
We experimented with multiple implementations. For example, only reducing the number of accesses to the input matrices. However, the best version is the one described above.

Vectorization. Vectorizing is straightforward. For each matrix we load 4 subsequent elements, do the computations and then store the result. The padé-coefficients are loaded into vector registers beforehand using the mm_set1_pd instruction. Moreover, we use fused multiply–add (FMA) instructions whenever possible. The addition of the diagonal matrix $\beta_0 \cdot I$ is done in a scalar loop as it cannot be vectorized efficiently. We implemented vectorized versions of the base implementation and the optimized scalar implementation.

3.6. Computing (3.6)

The equation (3.6) in [1] describes a system of linear equations. We computed the matrices Q_m and P_m using the equations (3.4) or (3.5) and our goal now is to find the matrix R_m that solves $Q_m * R_m = P_m$.
If the input matrix A is an upper or lower triangular matrix, then so is the matrix Q_m. In this case we can directly solve the given system using forward substitution or backward substitution respectively. Both substitution methods have a runtime of $O(n^3)$. Since the rescaling phase at the end of the matrix exponential algorithm is different for triangular matrices, we have to perform this check at some point. If we do so before the function (3.6) is called, we can make this case distinction without any additional cost.
In the case where Q_m is not triangular, there exist several different methods to obtain a solution to the given system of linear matrix equations. Probably the easiest and fastest one would be to use Gaussian elimination in combination with backward substitution. This method tends to be numerically unstable for ill conditioned matrices. Since we cannot guarantee the conditioning of the input matrices and checking the properties of the matrix would add an significant overhead to the cost, we chose to use a numerically stable algorithm right away.
For this reason we decided to use an LUP-decomposition algorithm in combination with forward and backward substitution. We chose to use only partial pivoting instead of full pivoting since it is computationally less expensive and is said to have feasible numerical stability in practice. Our tests have verified this assumption. This gives us the following structure for the function implementing equation (3.6):

if Q_m is lower triangular **then**
  $R_m \leftarrow forward\_substitution(Q_m, P_m)$
else if Q_m is upper triangular **then**
  $R_m \leftarrow backward\_substitution(Q_m, P_m)$
else
  $L, U, P \leftarrow LUP\_Decomposition(Q_m)$
  $P'_m \leftarrow mmm(P, P_m)$
  $Y \leftarrow forward\_substitution(L, P'_m)$
  $R_m \leftarrow backward\_substitution(U, Y)$
end if

LUP-decomposition. There are several different algorithms to compute the LUP-decomposition of a matrix. We chose the recursive-right-looking-algorithm since it seems to be the one that could benefit the most from using the optimizations we learned to use in the lecture e.g. unrolling and using SIMD-instructions. The algorithm has a runtime of $O(n^3)$ and performs the following steps:

1) Swap pivot-row with top-row

2) Partition matrix into Pivot element, vectors a_{12} and a_{21}, submatrix A_{22}

piv	a_{12}
a_{21}	A_{22}

3) Divide elements of a_{21} by pivot element

4) Compute the Schur's complement of A_{22}:

$$A'_{22} = A_{22} - (a_{21} * a_{12})$$

5) Continue at step 1) with matrix A'_{22}

Optimizations. In step 3) of the LUP-decomposition we have to divide all elements of the vector a_{21} by the pivot. To avoid the costly division, we can compute the reciprocal of the pivot and then multiply all elements of a_{21} with it. Since we use column-major format, the elements of a_{21} are aligned in memory. This enables us to use SIMD-instructions to perform the multiply on 4 elements simultaneously. In step 4) we have to subtract the outer product of two vectors from the sub-matrix A_{22}. This updating-step can be done using a double-loop with the following formula:

$A'_{22}[i][j] = A_{22}[i][j] - a_{21}[i] * a_{12}[j]$

As the sub-matrix A_{22} is stored in column major format, and the elements of the vector a_{21} are stored in order, we can use SIMD-instructions to update 4 elements simultaneously.

Forward-substitution. For a given lower-triangular Matrix L and a RHS Matrix B, we can find a solution Matrix Y by implementing the following formula using a triple-loop:

$y_{ij} = \frac{b_{ij} - \sum_{k=1}^{i-1} l_{ik} * y_{kj}}{a_{ii}}$

This formula shows that the computation of y_{ij} is dependent on $y_{(i-1)j}$. We therefore conclude that it is not possible to unroll the i-loop in a reasonable way.

Optimizations. The triple loop can either be implemented using an i-j-k loop order or using an j-i-k order. Both loop orders will access the matrix L in column-major order. When we access the h-th row of L, we use the first h columns of that row. The j-i-k order will access each row once before accessing it a second time. In total, each row will be accessed n-times. The i-j-k order will access each row n-times before continuing to the next row. Therefore the access pattern of the L matrix is more favorable when using the i-j-k loop order, but this loop order will access the matrices Y and B in row-major order. The j-i-k on the other hand will access the matrices Y and B in column-major order. Hence the j-i-k loop order is more favorable when not all 3 matrices fit into L1 cache. Additionally we have unrolled the j-loop and the k-loop by a factor of 4. This enabled us to use SIMD-instructions to perform the computation for 4 entries of Y simultaneously. The last optimization we performed is to avoid the costly division by computing the reciprocal of a_{ii} and then multiplying by it.

Backward substitution. The backward substitution works almost exactly the same as the forward substitution with difference that the order of the computation is reversed. We used the same optimizations and will therefore not explain it any further here to avoid redundancy.

4. EXPERIMENTAL RESULTS

In this section, we present the results of various experiments we did on the overall algorithm, but also on the sub-algorithms we presented in the previous section.

Experimental setup. For our benchmarks we used a Lenovo Thinkpad X1 Carbon 7th Gen.

- Processor: Intel Core i7 8565U (Whiskey Lake)
- Base Frequency: 1.8 GHz
- Max. Frequency: 4.2 GHz (Turboboost 2.0)
- Caches: L1: 64Kb, L2: 256Kb, L3: 8Mb
- GCC version: 13.1.1
- Flags used: -O3 -fno-tree-vectorize -ffp-contract=off -march=native -ffast-math

For our experiments we used dense matrices of sizes in the range from $n = 32$ to $n = 2048$. We executed our benchmarks with `taskset 0x1`, pinning the process to core 0 of the CPU.

4.1. Matrix Matrix Multiplication

For matrix matrix multiplication we show 4 different version. We assume all of them use $2 * n^3$ flops (OpenBlas as well).

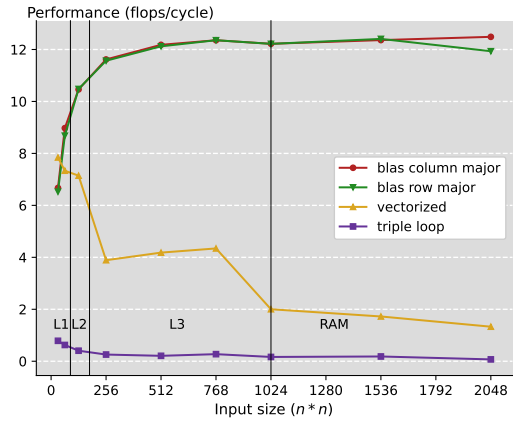

Fig. 2. Performance of four implementations of mmm used in our matrix exponential algorithm.

As depicted in Figure 2, the performance of our baseline implementation of the mmm algorithm (triple loop) drops depending on the input size of the matrices and the cache level they fit in.
Performance also drops for our vectorized version. Performance for matrices which fit in the L1 and L2 cache is

around 7 to 7.8 flops/cycle. As soon as the size is big enough to only fit in the L3 cache, the performance drops to around 4 flops/cycle and if they only fit in RAM, it drops to around 2 flops/cycle. The speedup, averaged over all results is 14.5.

OpenBlas performance is lower than the vectorized version only for the 32x32 matrix case. This is due to OpenBlas analyzing matrices before the actual mmm. As soon as a certain size is reached, performance peaks at around 12.4 flops/cycle, independent of cache level. It would not make sense to find a fix number for the speedup here because the bigger the matrix is, the bigger it will be. For example compared to our base implementation, the speedup for a 32 x 32 matrix is ˜8.5. and for a 2048 x 2048 matrix it is ˜175.8 which is a substantial range.

4.2. Matrix One Norm $||A||_1$

For the one norm function we benchmarked 4 versions. This algorithm uses $2 * n^2$ flops.

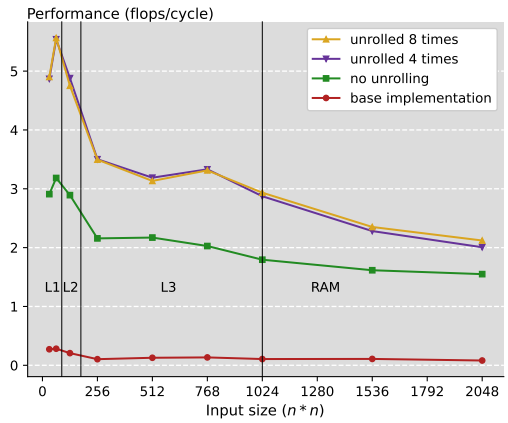

Fig. 3. Performance of four implemenations of onenorm, all except the base implementation are vectorized.

Figure 3 depicts the performance of our different versions of the one norm function. Again, performance drops with the cache levels. This plot also is a great example for the impact loop unrolling and ILP can have on such an algorithm. We found that unrolling the vectorized version 8 times does not make much of a difference compared to 4 times.
For a naively implemented, non unrolled vectorized version, the average speedup is around ˜15.5. For an unrolled version, it's about ˜24.

4.3. One Norm Estimation

We again have 4 versions for this algorithm. Because this algorithm is hard to analyze asymptotically, we counted the flops for this experiment with our flop count macro for each matrix individually.

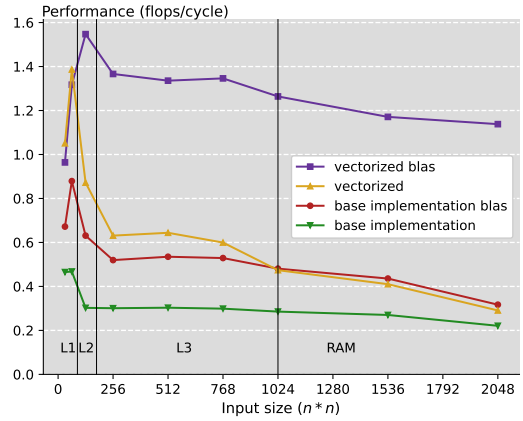

Fig. 4. Performance of four implementations of normest.

Performance of normest is relatively low compared to some other algorithms we optimized. This is due to a lot of overhead from load/store operations, integer operations and some calls to rand() in this algorithm.
Again, it is easy to see at which point matrices do not fit in cache anymore.
The speedup of the base implementation with blas compared to the vectorized version with blas ranges from ˜1.5 for smaller matrices to ˜3.0 for bigger matrices. The speedup for the versions without blas is around ˜2.0 as long as the input matrix fits in the cache. It then drops of to 1.32 for our biggest matrix. Compared to our base implementation without blas, the vectorized version with blas has a maximum speedup of ˜5.1. With bigger matrices this would probably get even higher.

4.4. Computing (3.4) and (3.5)

We benchmarked several versions but only show the most interesting ones, namely:

- Base implementation
- Vectorized base implementation
- Optimized, vectorized implementation
- As a comparison we show dgemm which is a single matrix-matrix multiplication using OpenBlas with a flop count of $2n^3$.

Results for (3.4). The benchmarks for (3.4) only consider the case $m = 9$ as this is the worst case scenario. It has a flop count of $2n^3 + 16n^2 + 2n$. As shown in figure 5, the vectorized versions beat the base implementation in all

cases and the optimized one performs better than the vectorized base. Apparently the trade-off between potentially fewer memory loads and conflict misses payed off.
The speedup of the optimization compared to the (scalar) base implementation is between 2.1 for small n and 1.05 for large n. Note that the runtime is dominated by the OpenBlas matrix-matrix multiplication. Hence, the speedup is bound to be relatively small, especially for large n where the mmm has a larger weight. We achieve 70% of OpenBlas performance for $n = 512$ and 95% for $n = 2048$.

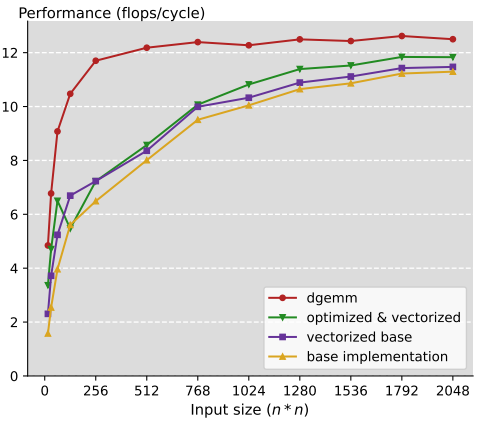

Fig. 5. Performance of three implementations of (3.4).

Results for (3.5). The flop count for (3.5) is $6n^3 + 24n^2 + 2n$. The results, shown in figure 6, are similar to (3.4). Our optimized, vectorized implementation performs better than the vectorized base implementation. The speedup is between 1.6 (for small n) and 1.03 (for large n) compared to the scalar base implementation. Our implementation achieves 83% of OpenBlas performance for $n = 512$ and 97% for $n = 2048$. This can again be explained by the fact that the mmm's dominate the performance, especially for large n.

Additional results for (3.4) and (3.5). We also experimented with unrolling. We show the results for the vectorized, optimized implementation with unrolling of 2 and 4 in the appendix A, Figures 10 and 11. While the unrolled versions perform slightly better in some cases, the difference is negligible.

4.5. Computing (3.6)

We tested several different approaches to solve the given system of linear matrix equations. In the end we chose to use an LUP-decomposition with the recursive-right-looking algorithm. Since algorithmic optimizations were not the

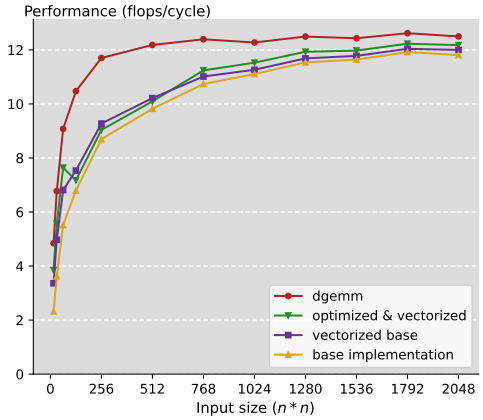

Fig. 6. Performance of three implementations of (3.5).

goal of this project we omit the results we obtained for different approaches and only present the results we got for our final choice. We tested 4 different versions of this approach. The base version and the vectorized version, each without using blas and with using blas. The results we obtained can be seen in Figure 7. As input we used non-triangular matrices since those required the most computational effort. The LUP-decomposition and both substitution functions have a flop count of $O(n^3)$. Adding the flop count of the matrix-matrix multiplication gives us a total flop count of $O(5n^3)$. For small matrices ($n \leq 128$) the maximum speedup we achieved was 12.7 without using blas and 4.4 with using blas. The respective minima were 8.2 and 2.4. For large matrices ($n \geq 1024$) the maximum speedup was 6.6 without using blas and 5.8 with using blas. The respective minima were 5.3 and 4.7.

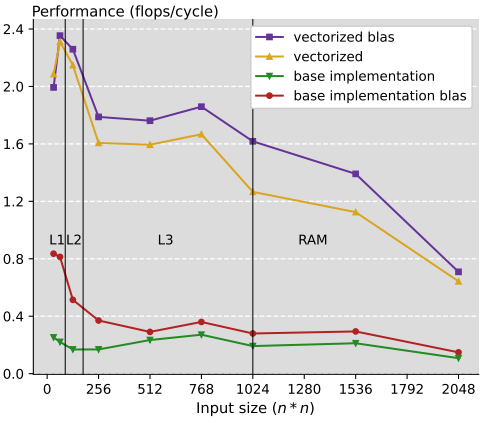

Fig. 7. Performance of four implementations of (3.6).

4.6. Matrix Exponential

This section contains the results of our experiments on the main algorithm. We only benchmarked the case where $m = 9$ because this is the worst case for which we can still use the padé approximants and provides the most interesting results. The flop count for the base and optimized versions is roughly the same, as measured by our flop count macros. The optimzed version uses a loop unrolling of 4.

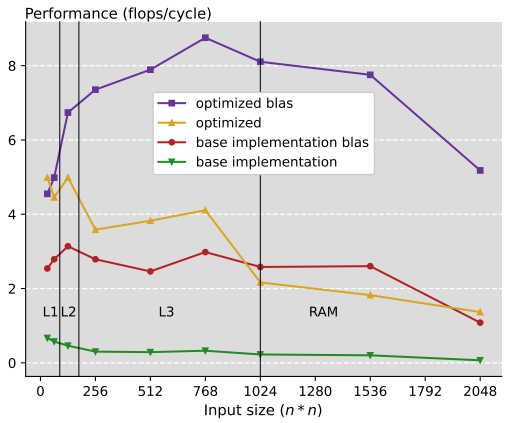

Fig. 8. Performance of four implementations of Matrix Exponential.

The main takeaway from figure 8 is that even with blas enabled, performance deteriorates with the sizes of the matrix. This is mostly due to the overhead from the LU decomposition in (3.6). We see the same effect in figure 7.
The plot also once again shows the impact of the size of the matrix on performance, especially the optimized version where OpenBLAS is not used illustrates that. The speedup from our optimized version compared to the base version on average is ˜11.4. For the versions using blas, it's ˜2.8.
Looking at the overall speedup from the base to the best version using OpenBlas for the mmm's, we find that due to OpenBLAS, the speedup continuously climbs from 6.8 for the 32x32 matrix to 75.7 for the 2048x2048 matrix.

Profiling. For profiling we used Intel Advisor. Figure 9 compares the base implementation to the vectorized version not using OpenBlas. The input matrix size is 512x512. The square dots are the results from the base implementation, the circular dots are from the optimized implementation. The size of the dots represents the time used inside these functions. Our main bottleneck is unsurprisingly the matrix matrix multiplication (If we do the same profiling with blas, mmm does not appear on the roofline plot anymore). Our second bottleneck is the LU decomposition and its forward and backward substitutions. This fits our explanation of the results we get from our performance plots.

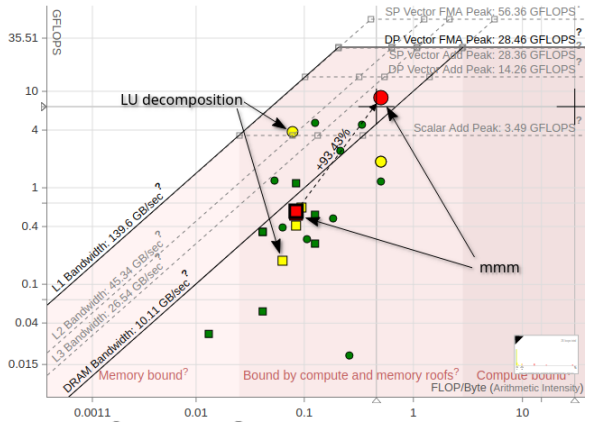

Fig. 9. Roofline Plot comparing our base implementation with our optimized implementation. Input size is $n = 512$

5. CONCLUSIONS

Our implementation is based on building blocks, which represent the functions we have described in our report. We believe that presenting these functions individually has given us a better understanding of how our optimizations have affected each building block. All of our functions have greatly benefited from our optimizations, but some of them have experienced a more significant increase in speed, even though we applied vectorization consistently.
For instance, the unrolled version of the one norm function has achieved a speedup of approximately 24x. This improvement can be attributed to the change in matrix format, as we now utilize the column major format. This shift has significantly enhanced the performance of this specific function and played a key role in our decision to adopt it.
Other parts of the algorithm like the functions (3.4) and (3.5) gained only a small increase in performance, when comparing only versions that use OpenBlas. The reason for this is that the performance is dominated by the OpenBlas mmm and therefore has little potential for optimization.
Our final matrix exponential algorithm has a speedup of around 2.8 compared to the baseline when we use OpenBlas in both cases. The main optimizations have been the switch to column-major format, vectorization and optimized memory access patterns.
In future work it would be interesting to optimize the unrolling factor for each loop individually. For the overall matrix exponential benchmarks we used the same amount of unrolling for all loops. Additionally, we could play more with different compilers and flags.

6. CONTRIBUTIONS OF TEAM MEMBERS

Andrea. Optimizations and analysis of (3.4) and (3.5), flop count macro.

Krishna. Optimization of mmadd (scalar multiplication and matrix addition), scalar matrix multiplication, matrix column sum and matrix absolute.

Philipp. Implementing and optimizing (3.6), code fragment (2.1) and some basic matrix operations.

Roman. Optimizations of matrix matrix multiplicatons, onenorm and normest. Benchmarks and profiling.

7. REFERENCES

[1] Awad H. Al-Mohy and Nicholas J. Higham, "A new scaling and squaring algorithm for the matrix exponential," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 970–989, 2010.

[2] I. Najfeld and T.F. Havel, "Derivatives of the matrix exponential and their computation," *Advances in Applied Mathematics*, vol. 16, no. 3, pp. 321–375, 1995.

[3] F. Tisseur N.J. Higham, "A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra," *Society for Industrial and Applied Mathematics*, 2000.

[4] The Pallets Projects, *Jinja*.

A. ADDITIONAL RESULTS FOR (3.4) AND (3.5)

We show some results for our best implementation of (3.4) and (3.5) with unrolling of 2 and 4, see Figures 10 and 11. In some cases, we observe a speedup but it is negligible. The reasons are that first of all the matrix-matrix multiplication dominates the runtime and the performance. Additionally, we are already using a lot of vector registers and doing a lot of computation in one loop iteration, some of them are independent of each other and can be interleaved.

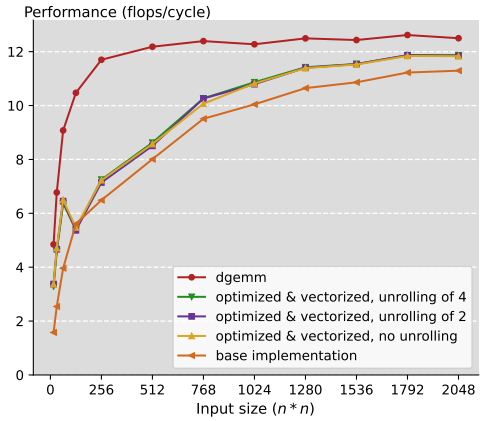

Fig. 10. Performance of unrolled, vectorized implementations of (3.4).

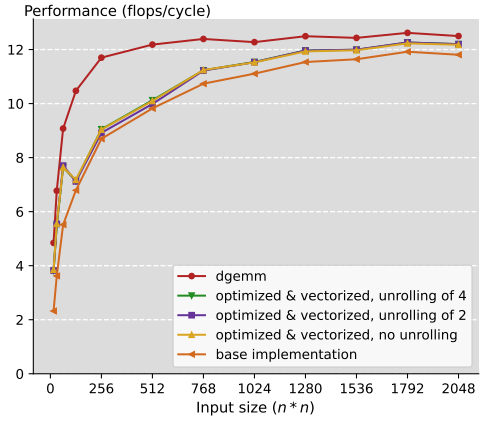

Fig. 11. Performance of unrolled, vectorized implementations of (3.5).