TOWARDS EFFICIENT MATRIX EXPONENTIAL COMPUTATION: A HIGH-PERFORMANCE C IMPLEMENTATION

Alejandro Cuadrón Lafuente, Antonino Orofino, Lorenzo Paleari, Julián Sainz Martínez

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper, we present an efficient implementation of Al-Mohy, Awad & Higham's algorithm for matrix exponential approximation in C. Our solution effectively addresses the primary computational bottlenecks, through optimization techniques such as blocking, matrix reutilization, and strength reduction. Results demonstrate our implementation's performance is comparable to Basic Linear Algebra Subprograms (BLAS) dgemm operations, reaching a peak performance of about 10 flops/cycle for 1024^2 matrices. On excluding dgemm operations, we observed a 7x performance improvement over the base implementation. We achieved a notable reduction in runtime, from approximately 5s to 850ms for matrices of size 1024^2. Our implementation competes with Scipy's `linalg.expm` up to matrices of size 512^2.

1. INTRODUCTION

Motivation. Matrix exponential computations are present in a variety of scientific and engineering disciplines, including but not limited to the resolution of linear ordinary and partial differential equations, quantum mechanics, control theory, and network analysis. The importance of efficient and accurate matrix exponential computations escalates with the rise in problem complexity and matrix sizes. This highlights the crucial need for fast and robust algorithmic solutions.

The commonly used approximation of the matrix exponential operation employs the scaling and squaring algorithm, as demonstrated by Higham's work, which forms the basis of MATLAB's `expm` function [1], the scaling and squaring method approximates the matrix exponential e^A as $(r_m(2^{-s}A))^{2^s}$, where $r_m(x)$ is the $[m/m]$ Padé approximant to e^x and integers m and s are chosen appropriately. However, this approach suffers from the deficiency known as overscaling, which involves selecting an unnecessarily large value for s, leading to accuracy losses in floating-point arithmetic.

The author thanks Jelena Kovacevic. This paper is a modified version of the template she used in her class.

To resolve this shortcoming, Al-Mohy, Awad & Higham introduced an innovative algorithm in their paper, "A New Scaling and Squaring Algorithm for the Matrix Exponential" that mitigates the overscaling problem, thereby enhancing both the accuracy and efficiency of matrix exponential computations [2]. This improved algorithm has been implemented in Python's well-known library, Scipy, as the `linalg.expm` method [3], affirming its applicability and significance. Yet, the pursuit of a fast implementation of this algorithm fully in C, despite its relevance, remains largely unexplored.

Contribution. The challenge of creating high performance implementations of such intricate algorithms is considerable, requiring the surmounting of multiple layers of complexity, including memory hierarchy considerations, utilization of vector instructions, and the improvement of Instruction Level Parallelism (ILP). Our strategy involved applying techniques such as strength reduction, improving spatial and temporal locality through loop reordering, loop unrolling, vectorization and blocking.

In this report, we introduce a high-speed implementation of Al-Mohy, Awad & Higham's matrix exponential approximation algorithm, leveraging the performance capabilities of the Basic Linear Algebra Subprograms (BLAS) and the Math Kernel Library (MKL) for matrix-matrix multiplications (gemm). Our primary objective is to deliver an implementation where the overall performance is on par with that of the gemm operations, while preserving the integrity of the original algorithm.

Related Work. There exists a body of work that includes the implementation of the algorithm proposed by Al-Mohy, Awad & Higham. Notably, MATLAB [4] and the Scipy library in Python [3] have implemented this algorithm. The optimization of this algorithm's implementation in C language is still a subject that requires further exploration.

The Scipy's `linalg.expm` implementation is written in Cython, a Python-C hybrid, which is expected to achieve performance levels similar to C. This makes it a relevant reference for performance comparison in our study. However, it's worth mentioning that our implementation is purely in C, complemented by the performance of matrix multiplication from MKL [5].

2. BACKGROUND ON THE ALGORITHM/APPLICATION

The matrix exponential, symbolised as e^A for any matrix A, is a central matrix function with widespread applications in diverse domains such as differential equations, graph theory, and network theory. It is characterised as the infinite series:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \frac{A^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{A^n}{n!} \quad (1)$$

Here, I denotes the identity matrix, A^n represents the matrix A raised to the power n, and $n!$ signifies the factorial of n. This series is convergent for all square matrices A.

The computation of the matrix exponential poses difficulties due to the intricate tasks of calculating higher powers of the matrix and factoring in the denominator. Classical techniques for computing the matrix exponential often engage the scaling and squaring method. This method scales down the matrix by a factor of 2^s for some s, computes the exponential of the scaled matrix using a Padé approximant, and subsequently squares the outcome s times.

A Padé approximant is a superior form of rational function approximation that provides a more accurate approximation to a function compared to a mere polynomial approximation. In particular, a Padé approximant of order $[m/n]$ is a rational function, the quotient of two polynomials of degree m and n respectively. It is the most effective approximation of a function by a rational function of a given order, as it matches as many derivatives as feasible at a given point.

Nevertheless, the selection of s is vital for the accuracy of the result. An excessively large s (overscaling) can lead to an erroneous computed exponential. This overscaling problem becomes more prominent for larger matrices, resulting in substantial errors in the computed matrix exponential. These issues are elaborately discussed in the paper "A New Scaling and Squaring Algorithm for the Matrix Exponential" by Awad H. Al-Mohy and Nicholas J. Higham [2].

In the forthcoming sections, we will illustrate Algorithm 5.1 from the aforementioned paper, which presents a novel approach to alleviate the overscaling problem. Prior to that, we will define *normest* as it is integrally used in the main algorithm.

Normest. The *normest* algorithm, also referred to as Algorithm 2.4[6], is a block algorithm designed to estimate the 1-norm of a matrix. The algorithm is characterised by a parameter t that can adjust the accuracy and reliability of the estimation, which is essentially a lower bound. The number of iterations and matrix products necessary for convergence is virtually independent of t.

The algorithm initiates with a starting matrix X. The first column of X is chosen to be the vector of 1s, which is the starting vector utilised in Algorithm 2.1. This feature ensures that for a matrix with non-negative elements, the algorithm converges with an exact estimate by the second iteration. The remaining columns are selected as random vectors, with a verification for and correction of parallel columns.

Subsequently, the algorithm iterates over the following steps:

1. Compute $Y = AX$.
2. Compute $g_j = \|Y(:,j)\|_1$ for $j = 1, 2, ..., t$.
3. Arrange g so that $g_1 \geq g_2 \geq ... \geq g_t$.
4. Let ind_{best} be the index j where $g_1 = \|Y(:,j)\|_1$.
5. Compute $S = sign(Y)$.
6. Compute $Z = A^T S$.
7. Compute $h_i = \|Z(i,:)\|_\infty$ for $i = 1, 2, ..., n$.
8. If $\max(h_i) \leq Z(:, ind_{best})^T X(:, ind_{best})$, then halt the iteration.
9. Arrange h so that $h_1 \geq h_2 \geq ... \geq h_t$ and reorder ind correspondingly.
10. Set $X(:,j) = e_{ind_j}$ for $j = 1, 2, ..., t$.

The algorithm ceases when the condition in step 8 is fulfilled. The final estimate of the 1-norm of the matrix is given by g_1.

For complex matrices, everything in this section remains applicable provided that sign(A) is redefined as the matrix $(a_{ij}/|a_{ij}|)$ (and sign(0) = 1).

Algorithm 5.1. We will now shift our focus to the algorithm itself: Algorithm 5.1, also known as the new scaling and squaring algorithm for the matrix exponential, evaluates the matrix exponential $X = e^A$ of $A \in C^{n \times n}$ via the scaling and squaring method. It is designed for IEEE double precision arithmetic.

The algorithm initiates by computing the square of the matrix A. It then computes the 1-norm estimate of A^2 and the maximum of the 1-norm estimate of A^2 and d_6. If certain conditions are met, it evaluates the polynomials $p_3(A)$

and $q_3(A)$ using Equation (3.4)[2] and verifies if the condition $|p_3(|A|)Te|_\infty / \min(|p_3|_1, |q_3|_1) \leq 10e^{\theta_3}$ is fulfilled. If the condition is satisfied, it evaluates the rational function r_3 using Equation (3.6)[2] and then halts the algorithm.

If the condition is not fulfilled, the algorithm proceeds to compute the fourth power of the matrix A and the fourth root of the 1-norm of A^4. It then computes the maximum of d_4 and d_6. If certain conditions are met, it evaluates the polynomials $p_5(A)$ and $q_5(A)$ using Equation (3.4)[2] and verifies if the condition $|p_5(|A|)Te|_\infty / \min(|p_5|_1, |q_5|_1) \leq 10e^{\theta_5}$ is fulfilled. If the condition is satisfied, it evaluates the rational function r_5 using Equation (3.6)[2] and then halts the algorithm.

The algorithm continues this pattern, computing higher powers of the matrix A, checking conditions, and evaluating polynomials and rational functions until it either halts or reaches the final steps where it scales down the matrices A, A^2, A^4, and A^6 by certain factors, evaluates the rational function r_{13} using Equations (3.5)[2] and (3.6)[2], and verifies if the matrix A is triangular. If A is triangular, it invokes Code Fragment 2.1[2]. Otherwise, it computes the matrix exponential X as $r_{13}(A)2^s$ via repeated squaring.

Cost Analysis. The cost of Algorithm 5.1 is compared with that of Algorithm 3.1 in the referenced paper. The authors display a series of graphs illustrating the ratio of the cost of Algorithm 5.1 to the cost of Algorithm 3.1 for varying test matrix sets.

The cost analysis discloses that Algorithm 5.1 is more efficient than Algorithm 3.1 in almost every instance, often significantly so. This increased efficiency primarily results from the exploitation of triangularity in the squaring phase of Algorithm 5.1.

However, the exact cost of Algorithm 5.1 is contingent upon the specific characteristics of the input matrix and the available computational resources. Therefore, the cost ratio presented in the paper should be considered a general indication of the relative efficiency of the two algorithms, not a precise measure of their costs.

Moreover, the computational cost of Algorithm 5.1 is expressed as $(\pi m + s)M + D$, where m is the degree of the Padé approximant used, πm is the cost of evaluating p_m and q_m (tabulated in [10, Table 2.2])[2], s is the scaling factor, M represents the cost of matrix multiplication, and D represents the cost of other operations.

If line 36 is executed, then the cost is calculated as $(\pi_{13} + s + 3)M + D$. If any of the tests at lines 5, 13, and 22 fail, then there is some wasted effort in evaluating lower-degree polynomials p_m and q_m that are not utilized.

The measurement of FLOPS for this project is now being explored. Only operations involving doubles were considered, and counters were utilized as a tool for quantification. For instance, every instance of using Blas for an MMM resulted in the addition of $2n^3$, with only the higher order being taken into account. This led to a FLOPS range between $6n^3$ and $28n^3$, inclusive of counts. Such oscillation can be attributed to the necessity of solving at least one system, and depending on the program's halting point, more or fewer MMMs will be computed.

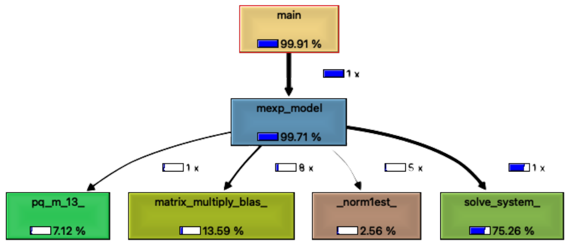

Fig. 1: callgrind cycle count graph with 1024 x 1024 matrix

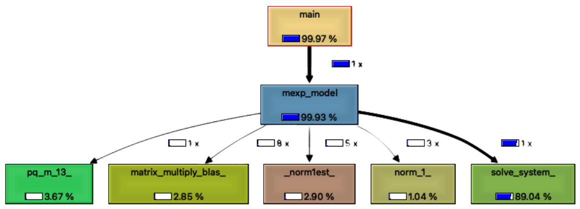

Fig. 2: callgrind cache miss count graph with 1024 x 1024 matrix

3. OPTIMIZATIONS

This section presents the optimization steps undertaken throughout the work, alongside with some analysis of the reasons behind the optimizations. These steps were crucial in achieving the final results. The initial phase of the project involved developing baseline implementations, which is introduced in the following paragraph.

Baseline Implementation. A baseline implementation was built following algorithm 6.1 [2], which is a refined version of algorithm 5.1 [2], both of which have been described in section 2. The algorithm make use of norm estimation [6] and super-diagonal exponential [7] which have been also implemented.

The vanilla C code doesn't feature any use of structure to represent matrixes that are contained in simple arrays of doubles. The code make use of the BLAS implementation of the *dgemm* function to perform MMM (matrix matrix multiplication) using the interface provided by MKL (math kernel library)[5].

First Analysis. To identify the most time-consuming functions we used a tool, *callgrind*[8], to profile the code and get insight on the functions performances.

The results of the analysis in Fig 1 highlight that the most time consuming function is *solve_system*.

Leveraging *callgrind* using `--simulate-cache=yes` option, we were able to have an estimation on the number of cache misses caused by each function. The results of this analysis is presented in Fig 2, where it can be observed that *solve_system* is responsible for the highest number of cache misses.

Leveraging callgrind the baseline implementation has been analyzed with various matrix sizes and inputs. Overall it can be observed that with small matrixes, the time spent in *solve_system* is still a major bottleneck, but since the number of cache misses is heavily reduced due to the fact that smaller matrices fits in Last Level Cache (LLC) it can be observed that also the time spent in *normest* became the second bottleneck.

After the analysis of the baseline implementation, we identified three major optimization points: *solve_system*, *normest* and the huge number of LLC misses when the matrices became too big.

Optimizations. Since the baseline code mainly access matrices in row major order making simple operations such as additions, substractions, scalar multiplications most of the optimizations can be directly applied to the code without much effort:

- **Instruction Level Parallelism.** Where possible the loop has been unrolled and accumulators have been used to allow the compiler to better exploit Instruction Level Parallelism (ILP), Write-After-Read (WAR) and Write-After-Write (WAW) dependencies have been removed by using temporary variables.

- **Removed procedure calls.** Expensive procedure code such has *abs()*, *pow()*, *ceil()*, *floor()* and *log()* have been removed, precomputed or replaced with cheaper operations depending on the context.

- **Precomputation.** In all possible cases the code has been modified to allow precomputation of values, such as divions or transposition and absolute of matrixes values.

- **Reuse of allocated space.** The algorithm need to store many intermediate matrixes containing partial results, one huge optimizations has been reusing a small set of allocated space for all the intermediate values necessary, reducing the number of cache misses and the number of memory allocations.

- **Blocking.** Before solving the matricial system up to 8 matrixes are used to compute the input matrixes for the system since all operations are in row major order, blocking has been easily applied allowing all operations performed on the same indexes to be processed togheter before moving on the next index. With bigger matrixes this optimization has been crucial in reducing the number of cache misses.

- **Vectorization.** Where applicable AVX2 intrinsics (Advanced Vector Extensions) have been used to vectorize additions, multiplications and substractions leveraging also the FMA (Fused Multiply Add) instruction. The loop has been unrolled to allow vectorization by a factor of 4 or 8 depending on the instruction type to leverage ILP.

Other optimizations such as **Inlining** and **Optimizations in *solve_system*** have been more difficult to apply:

- **Inlining.** The baseline implementation make extensive use of function calls, all code in each file was inserted and combined in a single C-file. Compiling with `-fopt-info-inline-missed` flag allowed us to observe which functions were not inlined by the compiler due to default behaviour and limitations of gcc. To force the compiler to inline every function we added to every function declaration `__attribute__((always_inline))`.

  While inlining the code has also been optimized to call few time at the beginning the *random()* procedure call, which is expensive, and then reuse the same random values for every *normest* call. Also it is possible to reuse computation performed in previous *normest* call to avoid recomputing the same values, ending up with a "scalar replacement" like optimization.

- **Optimizations in *solve_system*.** *Solve_system* use a pivoting implementation of the Gaussian elimination algorithm to solve the matrix system by inverting one input matrix and multiplying the inverted with the second input matrix. The algorithm has been optimized to avoid recomputing the same absolute values and to avoid divisions by pre-computing the inverse of the diagonal matrix.

  Further optimization like **vectorization** have been easily applied to the code, **blocking** has been tried but without success: the algorithm update the whole matrix at each step, so doing all the computation in a small block and so blocking is not possible because the algorithm need to access the whole matrix at each step to correctly update values and compute the next step.

Second Analysis and New Approach. Following the optimizations described above, we profiled the code again using *callgrind*[8] to verify the improvements.

The analysis showed that while overall the code has been improved, the number of LLC misses caused by *solve_sytem* remains unchanged and this cause the function to still be a major bottleneck.

To better visualize the improvements that reducing cache misses can bring, we analysed the code again using *Intel Advisor*[9], a state of the art tool that allow to visualize the number of cache misses and the number of cycles spent in each function combining the data into a roofline plot.

Leveraging the tool we observed that the number of LLC misses is still a major bottleneck and there's still huge room for improvement when matrices are too big to fit in LLC.

LU decomposition is a well known algorithm that can be used to solve a matrix equation of the form AX=B, despite having roughly the same computation complexity as Gaussian elimination ($O(n^3)$) can be implemented in a way that reduce the number of cache misses and so the number of cycles spent in the function.

The code has been reverted to base implementation and *solve_system* has been changed to use LU decomposition instead of Gaussian elimination. All optimized version created applying the optimizations described above have been maintained performing the same optimizations steps to LU decomposition. Into LU decomposizion we implemented wo major new optimizations, described in the next paragraph.

New optimizations. As stated above two new optimizations have been implemented into LU decomposition:

- **Loop order.** Loop order has been changed to improve spacial and temporal locality of the algorithm. It has been mostly effective in Forward and Backward substitution part of the algorithm, which are called many times during the execution of the algorithm. The code has been modified to go from:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < j; k++)
      P[j*n + i] -= P[k*n + i] * Q[j*n + k];
```

  to:

```
for (int j = 0; j < n; j++)
  for (int k = 0; k < j; k++)
    for (int i = 0; i < n; i++)
      P[j*n + i] -= P[k*n + i] * Q[j*n + k];
```

  The example refers to the Forward substitution part of the algorithm, a similar change has been applied to the Backward substitution part. Can be observed that the new loop order allow to access the same indexes of the matrices in the same order in the inner loop, improving spacial locality, and the same indexes of the matrices in the same order in the outer loop, improving temporal locality. The difficult part of loop order is to find the right order to improve locality without breaking the correctness of the algorithm that needs operations to be performed in a specific order.

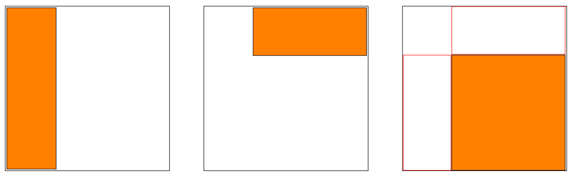
(a) Step 1, LU decomposition (b) Step 2, values update (c) Step 3, matrix multiplication

Fig. 3: Representation of blocking method used to reduce cache misses in LU decomposition.

- **Blocking.** The new algorithm has been implemented mainly due to the possibility to apply blocking to LU decomposition. We modified a naive implementation of the algorithm to follow three steps:

  The first step shown in Fig 3a consists in applying the standard LU decomposition algorithm to a set of column, the set consists of 128 columns for matrices of size 1024 x 1024 and 2048 x 2048. A more general approach would be to compute the number of columns to process in a set based on the size of the matrix and the size of the cache as follows.

  $$n\_columns = \frac{cache\_size}{sizeof(double)*matrix\_size*2}$$

  The second step shown in Fig 3b consists in updating the values of a set of $n\_rows = n\_columns$ that have not been processed in the first step. The update is performed using the values computed in the first step.

  The third step shown in Fig 3c consists in performing a matrix multiplication between some of the values computed in the first step and the values computed in the second step. In Fig 3c have been highlighted in red the values that are used in the matrix multiplication and in orange the values updated with the result of the matrix multiplication.

  The algorithm is then repeated in a smaller matrix represented by the orange area in Fig 3c until all the values of the matrix have been computed.

4. EXPERIMENTAL RESULTS

This section delves into the evaluation of our optimizations, comparing performance, runtime, and conducting a roofline analysis.

Experimental setup. All the experiments were conducted on an Intel Core i5-6400 Skylake CPU, running at 2700MHz, and supporting AVX2. Turboboost was disabled. Each core has a level 1 write-back 8-way set associative instruction cache of 32 KB, level 1 write-back 8-way set associative data cache of 32 KB and level 2 4-way set associative cache of 256 KB. A shared level 3 12-way set associative cache of 6 MB is available to all cores. The compiler used was GCC version 10.2.1. The compilation was performed using flags such as `-Wall -O3 -ffast-math -march=native -mfma -mavx2 -mavx`, chosen to optimize performance. The inputs to the algorithm were square matrices, with sizes ranging from $2^4 \times 2^4$ up to and including $2^{11} \times 2^{11}$, using size increments of power of 2. Experiments were stopped after a fixed number of iterations, depending on the experiment.

BLAS vs. NO BLAS. Our code extensively uses Matrix-Matrix Multiplication (MMM), and hence BLAS. We aimed to evaluate the relevance of our optimization steps compared to the BLAS performance. To achieve this, we created two versions for each step: one version includes the BLAS-performed MMM operations, while the other does not make use of BLAS and used precomputed matrices instead.

Branches analysis. As shown in section 2, the exact cost of the algorithm 5.1 heavily depends on the characteristics of the input values. Small values lead to the first branches of the algorithm being taken, while larger values cause the algorithm to execute till its end. Our benchmark infrastructure accounted for these possible cases and generated values accordingly to force each of the six possible branches. This approach allowed us to distinguish between these different cases and only compare performance of the same operation counts. Unless explicitly specified, all the plots presented in this section refer to the last return of the pseudocode. This choice is motivated by the fact that it allows more lines of the algorithm to be executed and hence more space to perform optimization.

Performance with GE. The performance of different implementations using the Gaussian elimination algorithm with and without BLAS are shown in Fig.4 and Fig.5 respectively. The black line in Fig. 4 indicates the performance of only MMM operations done using BLAS, used as a reference for the maximum possible values that we could have reached. In the same plot, we can see how the performance increases as going from the basic optimisations, through the in-lining and blocking, reaching the final optimisation step that consists of the use of vector instructions. However, for larger input sizes, the performance decreases, particularly when the matrices do not fit the Last Level Cache (LLC). This performance drop is influenced by the `solve_system` function that cannot be blocked.

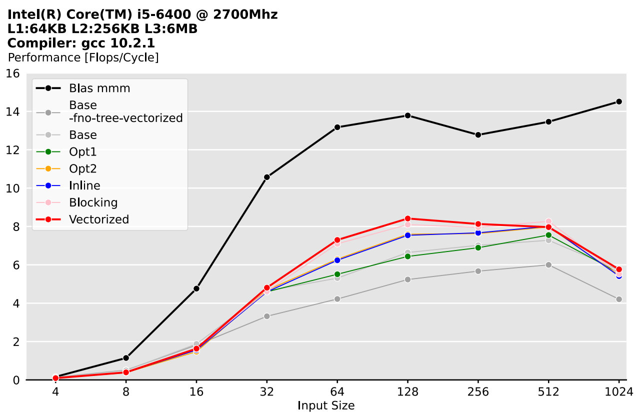

Fig. 4: Performance of six implementations of the algorithm 5.1 using Gaussian Elimination to solve the linear system and BLAS to perform MMMs. The operations count is roughly the same

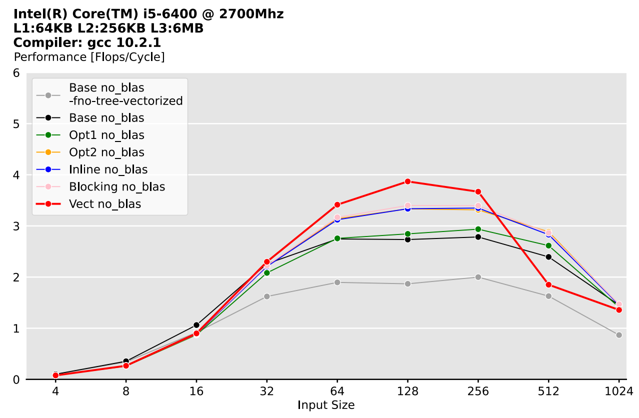

Fig. 5: Performance of six implementations of the algorithm 5.1 using Gaussian Elimination to solve the linear system and without making use of BLAS. The operations count is roughly the same

Moreover it must be noted that the drop in performance is more visible in Fig. 5 were BLAS do not boost the performance, BLAS is slightly more efficient with 1024^2 matrices and this cause the line to flatten and show less the drop in performance in Fig. 4.

We tried to use vector instructions as much as possible throughout the whole code, before the drop in performance we are able to achieve a 2x speedup over the baseline without vectorization enabled. The poor speedup obtained from vectorization is mostly due to the fact that the code is heavily memory bound and most of the time is spent in transferring data and not computing values.

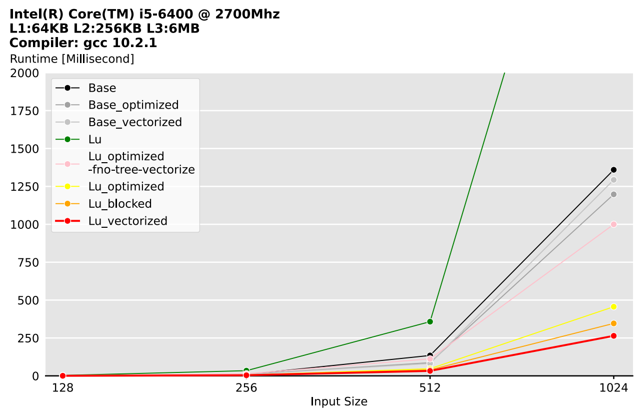

Fig. 6: Runtime comparison of different implementations of the two algorithm used to solve the linear system: Gaussian elimination and LU decomposition. Note that the green line reaches 4600ms, for better visualization the plot goes only from 0ms to 2000ms.

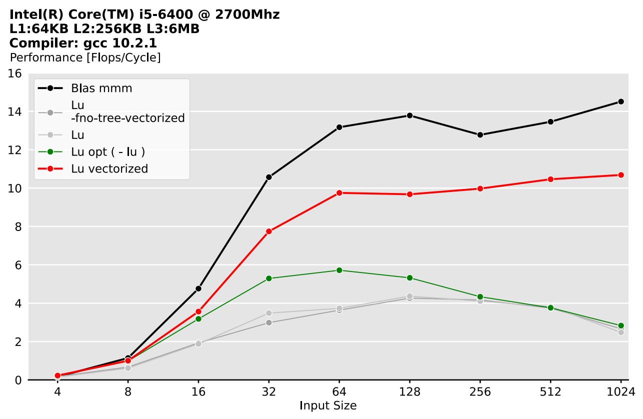

Fig. 7: Performance of six implementations of the algorithm 5.1 using LU decomposition to solve the linear system and BLAS to perform MMMs. The operations count is roughly the same

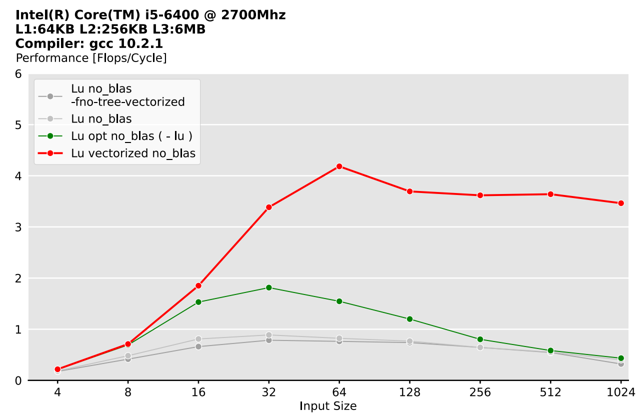

Fig. 8: Performance of six implementations of the algorithm 5.1 using LU decomposition to solve the linear system and without making use of BLAS. The operations count is roughly the same

Solving the linear system. This section seeks to address the question, "Is LU decomposition superior to Gaussian elimination?". As discussed in section 3, one of our optimization strategies involved identifying a more efficient algorithm to solve the linear system. Given the substantial difference in operation count between the two algorithms, we decided to compare the runtime of various optimization stages for each. As depicted in Fig. 6, while the base implementation of LU decomposition under-performs compared to the base implementation of Gaussian elimination, the vectorized version of LU decomposition significantly outperforms its Gaussian elimination counterpart. Specifically, it is 5.5 times faster for an input size of $n = 1024$. This observation underscores the effectiveness of LU decomposition when optimized with vectorization.

Notably the vectorization in LU decomposition is far more effective than previously with Gaussian Elimination, the pink line in Fig. 6 represent LU decomposition optimized with loop reordering techniques, but without vectorization enabled. Vectorization contribute with a speedup of almost 3.5x.

Performance with LU. This section aims to answer the question "How did the LU decomposition improve the performance of the algorithm?". The performance of different implementations using the LU decomposition algorithm with and without BLAS are shown in Fig.7 and Fig.8 respectively. Notably, the decreasing trend for larger sizes disappears when using LU decomposition, and in Fig. 7, we achieve a final performance close to our theoretical maximum represented by the BLAS line.

Can be also observed that vectorization, blocking and loop reordering on *solve_system* account for most of the speedup achieved: the green line represent all optimizations performed in the whole code but *solve_system*, while the red line adds also the optimizations performed on LU decomposition. Optimizations in the code are visible when input matrices are small, so when solving the system doesn't account for 90% of the runtime. It is worth noting that the red line achieve a 7x speedup, confirming the choice in LU decomposition and the effectiveness of all the optimizations implemented.

We were able to achieve 10 Flops/cycle compared to 14 Flops/cycle of BLAS, it is to be noted that for some input sizes the difference between us and BLAS is noticeably less.

Runtime comparison. This section aims to answer the question "How have all these optimization steps improved the runtime of the overall algorithm?". We achieved this by plotting one line for each version of our implementation to-

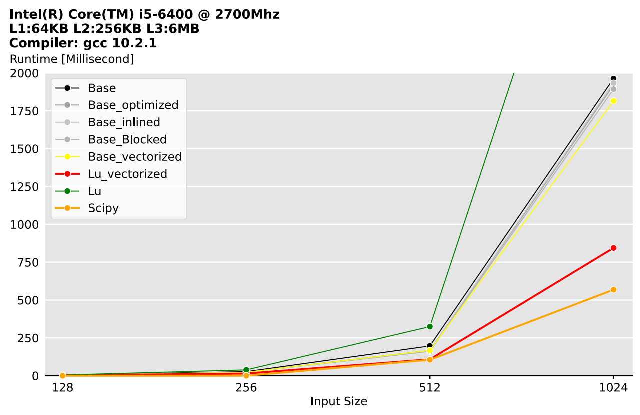

Fig. 9: Runtimes for the baselines, optimizations, as well as the known implementation. Note that the green line reaches 4600ms, for better visualization the plot goes only from 0ms to 2000ms.

gether with the runtime of the scipy implementation. We observe that until $n = 512$, we achieve the same runtime as Scipy. However, for larger sizes, our best implementation is outperformed by the scipy one. The scipy implementation uses BLAS also to solve the linear system, which we were not allowed to use. This could explain why scipy is faster than our implementation for larger sizes. We can also observe that the runtime differences between LU decomposition base implementation in green and our best version in red result in 7x speedup as mentioned before, going from 4.6s to 0.75s.

Roofline analysis. Roofline plots were constructed to compare the different improvements that were obtained after conducting the optimizations.
In this scenario, it is observed that the baseline LU (represented by the red square) was substantially inferior to the fully optimized result (the red circle) bounded by the L2 cache, and also less efficient than the non-vectorized version (the red triangle). Consequently, the enhancements applied to the baseline LU version demonstrated high efficacy, as the transition occurred from being compute-bound to memory-bound by the L2 cache. This roofline model was constructed using square matrices of size 512. Smaller improvements were observed in Gaussian elimination, as previously explained, leading to a decision to focus efforts primarily on LU decomposition. Hence, this is the reason why only this case is analyzed.

5. CONCLUSIONS

We accomplished significant optimizations, targeting the main bottleneck which involved solving large square matrix systems. Our strategy entailed transitioning from Gaussian elimination to LU decomposition, and then optimizing these and

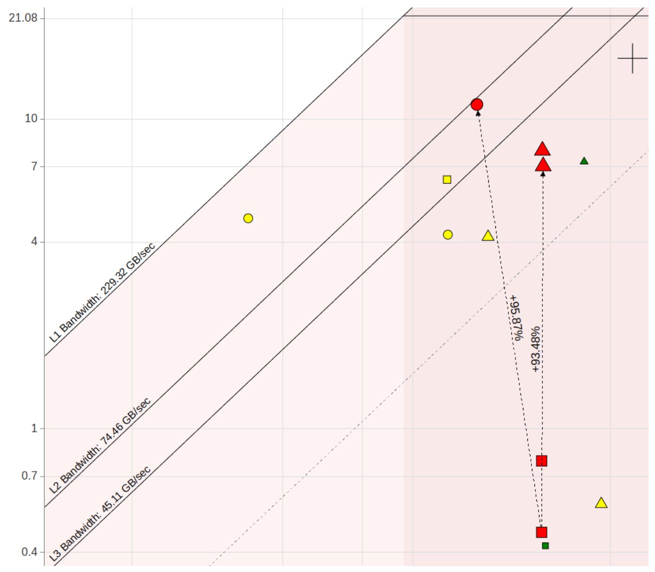

Fig. 10: Roofline plot for solve_system using LU

other operations through techniques such as loop reordering, blocking, matrix reutilization, and strength reduction. Our implementation demonstrated a performance that was comparable with that of the highly-optimized Basic Linear Algebra Subprograms (BLAS) gemm operations. Specifically, for square matrices of the size 1024^2, it achieved a peak performance of approximately 10 flops/cycle, while the BLAS gemm operations demonstrated a peak performance of about 14 flops/cycle. Interestingly, the performance of the BLAS gemm operation was so superior that it largely overshadowed the other optimizations we carried out. Upon excluding the gemm operations from our analysis, our implementation showed an impressive enhancement, with a 7x increase in performance over the base implementation. This highlighted the efficacy of our optimization strategies independently of the gemm operations. In terms of runtime, our base and final implementations showed a substantial improvement, from approximately 5s to 850ms, for matrices of size 1024^2. Our implementation demonstrated performance comparable to the *Scipy* implementation up to square matrices of size 512^2. However, for larger matrices, the scipy version, which leverages the LAPACK system solver, proved to be more efficient, achieving a runtime of approximately 500ms for matrices of size 1024^2.
Future work. may include the use of LAPACK system solver function, further fine tuning for block size and exploring vectorization with AVX512.

6. CONTRIBUTIONS OF TEAM MEMBERS

Alejandro Cuadrón Lafuente. Normest was designed and coded according to the specified algorithm. The design and coding of the main algorithm was carried out collaboratively with Lorenzo Paleari. All non-SIMD optimizations in Normest were conducted, and the integration of it with the primary function was done. Parts of the main algorithm were re-optimized using non-SIMD techniques. Normest computations were distributed throughout the algorithm to reuse earlier calculations. The roofline plots were crafted, and assistance was provided in the development of the presentation.

Antonino Orofino. Coded some of the utils functions and contributed to their initial optimisation and vectorization trial which consisted of strength reduction and increased the instruction level parallelism. Performed matrix transpose in order to optimise locality in the Gaussian Elimination algorithm. Contributed in the development of the forward and backward substitution of solve system with LU factorisation. Developed the cost metric and instructed the code to do the flop count through compiler macros. Collaborated on the choice of compiler flags and experiments. Used callgrind to identify and analyse the bottlenecks of the base implementation. Contributed on the final presentation including the parts that were related to my work.

Lorenzo Paleari. Implemented second half of the basic implementation of the main algorithm. Worked on basic optimizations mainly on main algorithm, which consists of strength reduction, Instruction Level Parallelism implementation, loop unroll and scalar replacement, precomputation of division and common values and reuse of allocated space. Collaborated with Alejandro Cuadrón Lafuente inlining *normest* into the main code. Performed blocking on main algorithm and contributed in blocking LU decomposition. Implemented vectorization on the main algorithm and helped implementing it in *solve_system*. Created performance and run-time plots for presentation and report.

Julián Sainz Martínez. Did first half of the basic implementation of the main algorithm. Coded part of the auxiliary functions in the utils file. Implemented the code for matrix inversion using Gaussian Elimination. Worked on some optimisations on the solve system code, which consisted of strength reduction, loop unrolling and minimal vectorisation. Performed also some initial vectorisation of the rest of the functions in utils (most of them were already done by the compiler). Developed the new method for solving the linear system based on LU decomposition and performed optimisations based on strength reduction and loop reordering to improve local and spatial locality. Helped with the slides regarding these parts in the final presentation.

7. REFERENCES

[1] Nicholas J Higham, "The scaling and squaring method for the matrix exponential revisited," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, no. 4, pp. 1179–1193, 2005.

[2] 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.

[3] The SciPy community, "Scipy v1.8.0 reference guide: scipy.linalg.expm," 2023, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.expm.html`.

[4] "expm in matlab," `https://it.mathworks.com/help/matlab/ref/expm.html`, Accessed: 2023-06-21.

[5] "Intel math kernel library," `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`, Accessed: 2023-06-21.

[6] Nicholas J. Higham and Françoise Tisseur, "A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra," *SIAM Journal on Matrix Analysis and Applications*, vol. 21, no. 4, pp. 1185–1201, 2000.

[7] Nicholas J. Higham, *Functions of Matrices: Theory and Computation*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2008.

[8] The Valgrind developers, "Valgrind: Callgrind," 2023, `https://www.valgrind.org/docs/manual/cl-manual.html`.

[9] Intel, "Intel advisor," 2023, `https://software.intel.com/content/www/us/en/develop/tools/advisor.html`.