Implementation Project
Modèles de Calcul

Jan Esquivel Marxen, Thomas Gantz, Laura Giò Paxton

course: MU4IN901

Contents

1 Project Description

This C code implementation addresses several key matrix operations and algorithms, each tested on matrices of varying sizes with uniformly distributed random floating-point coefficients in [-1, 1]. The tasks include:

1. Naive Matrix Multiplication: Implementing the classical algorithm for multiplying two matrices using double precision floating point numbers.
2. Strassen Matrix Multiplication: Implementing the Strassen algorithm for matrix multiplication using double-precision floating-point numbers.
3. Naive LU Decomposition: Developing a classical LU decomposition method for a matrix using double precision floating point numbers.
4. Matrix Inversion Using LU Decomposition: Using the LU decomposition function to compute the inverse of a matrix.
5. Strassen Matrix Inversion: Implementing two versions of the Strassen algorithm for inverting a matrix:
   - One version based on naive matrix multiplication.
   - Another version based on Strassen matrix multiplication.
6. Comparative Analysis: Performing a thorough comparison of the various implementations for matrix multiplication and inversion to compare their performance against one another and implementation efficiency, comparing their empirical-taken time against their theoretical time complexity.

2 Project Layout

The entire project is distributed in a standard Cmake project, with modules' header files `.h` in the *include* directory and the corresponding source code `.c` in the *src* directory.

The `.c` files contain 1-2 main functions that solve the problems described in the previous section. In addition, they contain helper functions that aid the main functions. These solve sub-problems or side-problems that may come up more often than once, and their being defined separately allows for better function structuring.

The files `naive_matmat`, `strassen_matmat`, `naive_lu` and `strassen_inv` include functions to solve, as their names suggest, the main problems described in section 1.

The two files `block_utilities` and `IO` contain utility functions, abstracted from their usage in different contexts, such as a simple matrix addition or printing matrices that are needed in more than occasion.

Finally, we have the file `test.c` that tests correctness of the main functions of different implementations of the matrix operations against a ground truth using the libraries `LAPACKE` and `OPENBLAS` and makes use of the `clock()` function in order to ultimately compare the running times of each implementation.

The `main.c` randomly generates input matrices such that all test functions of the same matrix operation are called on the same input data to facilitate the comparison of methods. The running times can then be compared. Noteworthy is that the randomly generated matrix for the inversion test function is first checked for invertibility. If it is not invertible, a new random matrix is generated.

A note to be made is that the code does not do error handling. What is meant by this is the code is not robust to miss-use, e.g. wrongly formatted input data.

The files or programs that contain the implementation of the matrix operations will be analyzed in more detail in the following sections.

3 Matrix Multiplication

3.1 Naive Matrix Multiplication

Program Description

The program is an implementation of the naive matrix multiplication. The meaning of a naive implementation is that the structure of the algorithm follows the simplest algorithm, such as would a human do when solving a matrix multiplication.

Input-Output Behaviour

The input of the function consists of three matrices, followed by their sizes. The input matrices are of type `double *matrix`, hence a one-dimensional array of double precision floats. The sizes are of type `const size_t n`. The first matrix, matrix `*A`, is of size `m` $\times$ `n` and the second matrix `*B` is of size `n` $\times$ `k`. The last matrix `*C` is of size `m` $\times$ `k` as it is the multiplication of `*A` and `*B`.

There is no output as the function is a void function. Yet, within the function the matrix `*C` will be updated and hence changed.

Program Design

The sizes `m` and `k` determine the first two `for` loops and the value `n` determines the last `for` loop which iterates over the $\texttt{m}^{th}$ row of `*A` and the $\texttt{k}^{th}$ column of `*B` to determine the element `*C[m][k]`. The matrix `*C` is passed as a pointer to the function and updated element by element.

The file contains the header file `#include <stddef.h>` for `size_t`. Otherwise no auxiliary functions are needed.

Design Choices

Matrices are implemented as one-dimensional arrays of doubles instead of two-dimensional arrays (pointer-to-pointer representation). This choice improves cache efficiency, as data is stored in contiguous memory, leading to faster access times for large matrices due to spatial locality. It also simplifies allocation, requiring only a single call to `malloc` instead of multiple allocations for rows and row pointers, thereby reducing overhead and fragmentation risks. Additionally, this format is compatible with libraries and APIs such as `OPENBLAS` and `LAPACKE`, allowing direct use without extra transformations.

The use of `const size_t m` ensures that matrix sizes remain constant and cannot be accidentally altered during execution.

These design choices hold throughout the rest of the project.

Complexity

The outermost loop iterates over `m` (the rows of `*A`). The second loop iterates over `k` (the columns of matrix `*B`). The innermost loop performs `n` multiplications and additions for each element of the resulting matrix `*C`.

Thus, the total number of operations is proportional to $m \cdot k \cdot n$, resulting in an arithmetic complexity of $\mathcal{O}(m \cdot n \cdot k)$.

3.2 Strassen Matrix Multiplication

Program Description

This C module implements Strassen's matrix multiplication. One can refer to the following lecture notes to find a detailed description of the algorithm [JBD]. Its goal is to improve the complexity of the matrix multiplication from $\mathcal{O}\left(n^3\right)$ to $\mathcal{O}\left(n^{2.81}\right)$ by improving the base case of the 2×2 matrix multiplication, then using a divide and conquer strategy by recursion.

Input-Output Behaviour

The Input-Output behaviour is very similar to that of the naive matrix multiplication implementation.

The input of the function consists of three matrices, followed by their sizes. The input matrices are of type `double *matrix`, hence a one-dimensional array of double precision floats. The sizes are of type

`size_t n`. The difference to the naive implementation is that the sizes are no longer constant given that they may change throughout the program, due to the necessity of padding. They will be reverted to their original size by the end of the function.

There is no output as the function is a void function. Yet, within the function the matrix `*C` will be updated and hence changed.

Program Design

The function is recursive, segmenting the matrices into smaller blocks and performing the Strassen matrix multiplication on the smaller blocks when appropriate according to the algorithm.

The code starts by determining whether a base case has been met. This ensures the correct implementation of the recursive nature of the code as it will return once a base case has been met. The base cases are either when all sizes `m, n, k` $<$ `512` where a naive matrix multiplication is performed or two of the matrix dimensions are one, for then a straightforward scalar vector multiplication can be performed.

From there the recursion process starts, meaning the matrices `*A, *B` and `*C` are firstly padded. Then, as the algorithm suggests, the matrices are segmented with matrix additions or recursive Strassen matrix multiplications performed on them. The partial results are then added in the correct position to the final `*C`. `*A, *B` and `*C` are then de-padded.

The modules `block_utilities` and `naive_matmat` are required as the naive matrix multiplication function is used as are matrix block manipulations.

Two auxiliary functions that are needed are the `pad_matrix` and `depad_matrix` functions. The reason for needing the padding function is that the Strassen algorithm must be performed on matrices that have a size divisible by 2. They copy the memory of the original matrix and paste it into a new memory location with a new size. The padding function increases the size by one and fills the new row or column with zeros. Whereas the de-padding function decreases the matrix to the original size before the padding.

Design Choices

A design choice that was made for this function was having pointers to pointers to the memory address of the matrices as an input. This was due to the fact that we needed to pad and de-pad the matrices and have the entries of the padded matrices have contiguous memory. Since these functions change the memory address of the matrices, yet pointers passed through the function as the input, eg. `**C`, should remain consistent in order to be correctly updated, one requires a pointer to a pointer to ensure both are possible. This way the initial first level pointer to the pointer of the address, `**C`, stays consistent but the second level pointer, `*C`, may change.

A second design choice is that of the first base case. The code takes `m, n, k` $<$ `512` to be a case where the naive matrix multiplication function is called. The reason for this base to be chosen is that the Strassen algorithm only becomes more efficient after the matrix dimensions exceed 512 [HSHvdG16]. Hence, to avoid matrix subdivisions, memory allocations and additional function calls (which would be necessary if one recursively called the Strassen matrix multiplication on the base case), the naive matrix multiplication function is called once.

Complexity

The arithmetic complexity of the provided code is primarily determined by the use of Strassen's algorithm for matrix multiplication. The key components of the complexity are:

- **Padding:** The `pad_matrix` function introduces overhead for copying matrices when padding is required, with complexity:
$$\mathcal{O}(mn + nk + mk)$$
where m, n, and k are the matrix dimensions.

- **Strassen's Recursive Multiplication:** The recursive relation for Strassen's algorithm is:
$$T(m, n, k) = 7 \cdot T\left(\frac{m}{2}, \frac{n}{2}, \frac{k}{2}\right) + \mathcal{O}(mn + nk + mk).$$
Solving this recurrence gives by the Master Theorem an overall complexity of:
$$\mathcal{O}\left(m^{\log_2 7} + n^{\log_2 7} + k^{\log_2 7}\right),$$

where $\log_2 7 \approx 2.81$.

- **Base Case:** For small matrices ($m, n, k < 512$), the algorithm falls back to naive matrix multiplication with complexity:

$$\mathcal{O}(m \cdot n \cdot k).$$

In summary, the overall complexity for large matrices is determined by Strassen's algorithm:

$$\mathcal{O}(max\{m, n, k\}^{2.81}).$$

4 Matrix Inversion

4.1 Inversion by LU Decomposition

Program Description

The goal is to invert a matrix and therefore solve n linear systems using the LU decomposition. The module `lu_naive` implements the LU decomposition (function `lu_decomposition()`) and matrix inversion (function `lu_invert()`). LU decomposition transforms a square matrix $\mathbf{A}$ into two matrices: a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, such that:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$$

where $\mathbf{L}$ has $l_{ii} = 1, i \in 1, ..., n$.

Input-Output Behaviour

The input consists of:

- Matrix $\mathbf{A}$, provided as an array of type `double *`, representing a square matrix of size $n \times n$.
- Matrix **inverse_A**, provided as an array of type `double *`, representing the inverse of $\mathbf{A}$ of size $n \times n$. Initially filled with invalid values, the result will be stored in this array.
- The matrix size n, given as a constant of type `const size_t`.
- There is no explicit output as the function is one of return type `void`.

The output is the result of the matrix inversion which is stored in the output matrix **inverse_A**.

Program Design

The program consists of two core components:

1. **LU Decomposition:** The function `lu_decomposition()` applies Gaussian elimination to transform $\mathbf{A}$ into its LU form. The matrix $\mathbf{T}$ stores both the lower triangular $\mathbf{L}$ and upper triangular $\mathbf{U}$ components.
2. **Matrix Inversion:** Then, the function `lu_invert()` solves for each column of the inverse matrix $\mathbf{A}\mathbf{A}_i^{-1} = \mathbf{L}\mathbf{U}\mathbf{A}_i^{-1} = \mathbf{L}\mathbf{y} = \mathbf{e_i}$ by:
   - Performing forward substitution on $\mathbf{L}$ (lower part of ($\mathbf{T}$) to solve for an intermediate vector $\mathbf{y} = \mathbf{U}\mathbf{A}_i^{-1}$.
   - Performing backward substitution on $\mathbf{U}$ (upper part of ($\mathbf{T}$) to compute each column of the inverse matrix $\mathbf{A}_i^{-1}$.

where $\mathbf{A}_i^{-1}$ is the i-th column vector of the inverse $\mathbf{A}^{-1}$ and $\mathbf{e_i}$ the i-th canonical vector.

Complexity

LU decomposition requires Gaussian elimination: The process involves pivoting over each column j (from $j = 1$ to n) and eliminating entries below the diagonal in that column. For each pivot column j, the number of row operations is $n - j$. When summing these operations across all n columns, the total number of operations can be expressed as

$$\sum_{j=1}^{n}(n - j) = \frac{n(n-1)}{2}$$

so total complexity is $\mathcal{O}(n^3)$ for an $n \times n$ matrix.
Matrix inversion also involves solving n linear systems, each requiring forward and backward substitution $\mathcal{O}(n^2)$, resulting in an overall complexity of $\mathcal{O}(n^3)$.

4.2 Strassen Matrix Inversion Using Naive Matrix Multiplication

Program Description

The function `strassen_invert_naive_matmat()` in the module `strassen_invert` implements the Strassen inversion algorithm applying the naive matrix multiplication. The recursive approach divides the matrix into four sub-blocks and solves for the inverse using a series of block operations, including matrix addition, multiplication, and inversion itself. If the matrix size n is not a power of 2, the algorithm pads the matrix to the nearest size that is divisible by 2, so that we can further divide into even blocks.

Input-Output Behaviour

The input consists of:

- Matrix $\mathbf{A}$, provided as an array of type `double *`, representing a square matrix of size $n \times n$.
- Matrix $\mathbf{A}^{-1}$, provided as an array of type `double *`, which will store the computed inverse of $\mathbf{A}$.
- The size n of the matrix, given as a constant of type `const size_t`.
- Again, the function has no explicit output.

Program Design

The program follows a recursive divide-and-conquer approach to compute the inverse matrix:

1. **Matrix Padding:** If the input matrix size n is not a power of 2, it is padded to $n_{\text{new}} \times n_{\text{new}}$, where n_{new} is divisible by 2. The new diagonal padding elements are initialized to 1, while other new elements are set to 0, since to compute the inverse of
   $\mathbf{A}_{\text{pad}} = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$
   we observe that
   $\mathbf{A}_{\text{pad}}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$.
2. **Block Decomposition:** The matrix $\mathbf{A}$ is split into four submatrices:
   $$\mathbf{A} = \begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{bmatrix}.$$
3. **Recursive Inversion:**
   (a) Compute $e = a^{-1}$ recursively.
   (b) Compute $Z = d - ceb$.
   (c) Compute $t = Z^{-1}$ recursively.
   (d) Compute $y = -ebt, z = -tce, x = e + ebtce$.
   (e) Return $\begin{bmatrix} x & y \\ z & t \end{bmatrix}$.

   To do these block computations we use temporary variables to store intermediate results, the function `naive_matmat()` from the `naive_matmat` module for matrix multiplication and the functions `darray_add()` and `mat_inplace_block_add()` from the `block_utilities` module adding blocks into $\mathbf{A}$.
4. **Matrix Depadding:** If padding was applied, the resulting matrix is depadded to its original size.

Complexity

The program follows a recursive divide-and-conquer strategy to compute the inverse of a matrix.

Matrix Padding: If the input matrix size n is not a power of 2, it is padded to $n_{\text{new}} \times n_{\text{new}}$ (the nearest power of 2). Padding involves memory allocation and initialization, with complexity $\mathcal{O}(n_{\text{new}}^2)$.

Block Decomposition and Recursive Inversion: The input matrix is split into four submatrices of size $\frac{n_{\text{new}}}{2} \times \frac{n_{\text{new}}}{2}$. Since $n \leq n_{\text{new}} \leq n+1$ we can limit our complexity analysis to n. Recursive steps are:

- Compute $e = a^{-1}$ recursively, contributing $T(n/2)$, where T is the time to compute the inverse of a matrix of size n.
- Compute $t = Z^{-1}$ recursively, contributing $T(n/2)$.
- Compute matrix additions and multiplications during each step $\mathcal{O}(n^3)$.

The time complexity can thus be expressed with the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n^3),$$

where:

- $T(n)$: Time to compute the inverse of a matrix of size n,
- $2T(n/2)$: Two recursive calls on submatrices of size $\frac{n}{2} \times \frac{n}{2}$,
- $\mathcal{O}(n^3)$: The asymptotic cost of matrix multiplications and additions.

Using the **Master Theorem** [JBD] on the recurrence of type $T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}(n^d)$, we have $a = 2$, $b = 2$, and $d = 3$. We compute $b^d = 2^3 = 8$. Since $2 < 8$, by the Master Theorem, the polynomial matrix multiplication dominates the recursion depth. Therefore:

$$T(n) = \mathcal{O}(n^3).$$

Thus, the overall complexity of the Strassen inversion algorithm is $\mathcal{O}(n^3)$. In general, the matrix multiplication algorithm dictates the algorithm's complexity, i.e. the matrix inversion is no harder than matrix multiplication [CLRS09].

4.3 Strassen Matrix Inversion Using Strassen Matrix Multiplication

Program Description

The function `strassen_invert_strassen_matmat()` in the module `strassen_invert` implements Strassen inversion integrating Strassen matrix multiplication. The recursive approach divides the matrix into four sub-blocks and solves for the inverse using a series of block operations, including matrix addition, multiplication, and inversion.

The algorithm solves for the inverse of a square matrix $\mathbf{A}$ of size $n \times n$, producing the resulting inverse matrix $\mathbf{A}^{-1}$.

Input-Output Behaviour

Exactly the same as in 4.2.

Program Design

Same as in 4.2, except that now at each recursion step strassen (instead of naive) multiplication is used if the dimensions of the submatrices are big enough ($n > 512$).

Complexity

The complexity of Strassen's matrix multiplication is approximately $\mathcal{O}(n^{2.81})$. Using the same reasoning (Master Theorem) as in 4.2, we see that this algorithm has complexity $\mathcal{O}(n^{2.81})$. This is an improvement over the naive multiplication and LU inversion with $\mathcal{O}(n^3)$.

The Strassen inversion yields the same asymptotic complexity as the integrated multiplication algorithm. This is why using the Strassen multiplication turns to be more performant than using the naive multiplication approach.

5 Test Setup

5.1 Test Function

Program Description

The goal is to test our implementation. To achieve this, we design a homogeneous, reusable, and randomised test interface. The test functions shall provide an accessible entry point to verify the correctness and efficiency of our implementation without requiring a detailed understanding of its internals, comparing the empirical-taken time against the theoretical time complexity.

Input-Output Behaviour

The input to a test function can include a matrix to be inverted or two matrices to be multiplied, along with their respective sizes and the accuracy required to match a ground truth result. The output is either `-1`, indicating an incorrect result, or the time taken to perform the function under test.

Program Design

There is a dedicated test function for each major functionality, adhering to the naming convention `test_<function_name>`. Each test function follows these steps:

1. Compute the ground-truth.
2. Execute our implementation while measuring the execution time.
3. Compare the results coefficient-wise.

Design Choices

Ground truth results are obtained using the `OPENBLAS` and `LAPACKE` libraries. As the location of the data may change during the execution of the function, the data must be passed as a double pointer in these cases.

5.2 Main

Program Description

The `main` function shall orchestrate the execution of all test functions for various input sizes and random matrices with coefficients in $[-1, 1]$ and provide a formatted output of the test results.

Input-Output Behaviour

The user can specify a maximum test size (`N`), corresponding to matrix sizes of approximately 2^N, when running the script. By default, `N` is set to 5. The results of all tests are displayed in the console and stored in files for further analysis and plotting.

Program Design

The main function executes the following sequence, first for matrix multiplication and then for matrix inversion:

1. Initialise random test data.
2. Call the respective test functions, flushing the cache in between.
3. Print results to the console and store them in files.

During testing, we observed varying timing results depending on the order of execution. To address this, a `flush_cache()` function was implemented to ensure no residual data remains in the cache, which could bias timing measurements.

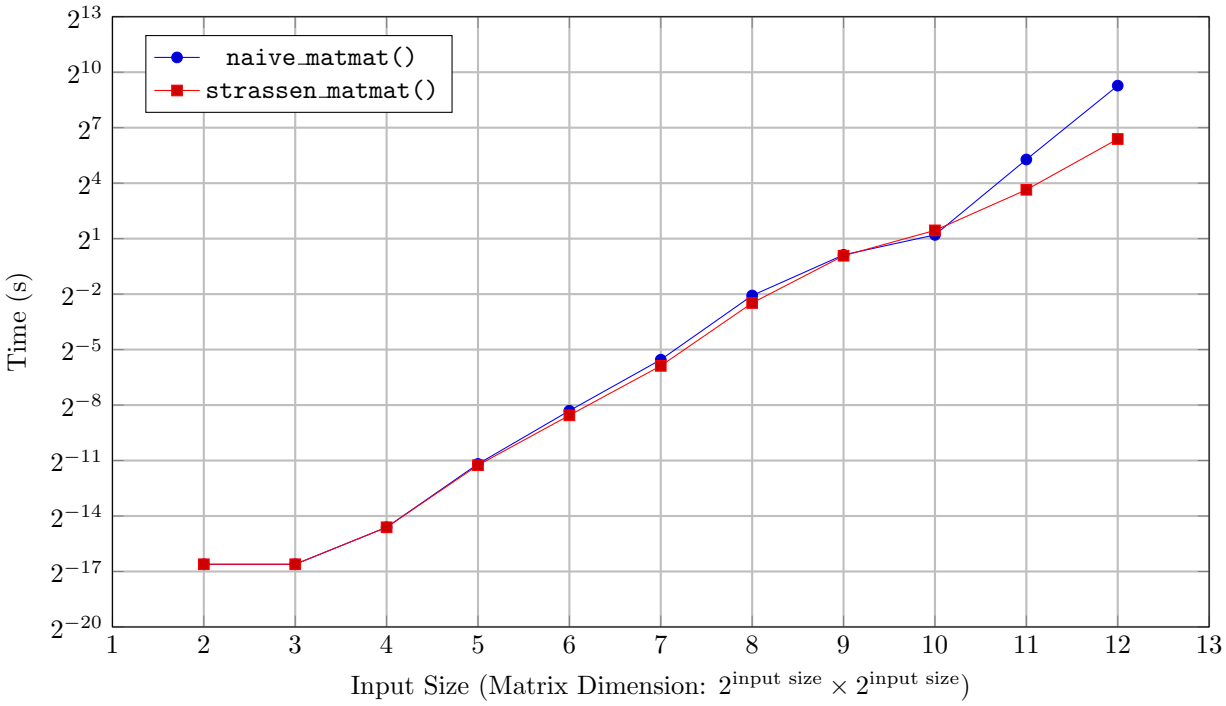
Design Choices

For both scenarios, random test data is generated, and the respective test functions are called for each implementation using the same test data to ensure comparability. Matrix sizes are chosen to slightly deviate from powers of two. This approach allows testing of padding scenarios while maintaining a consistent factor-of-two increase in size.

6 Comparison of the Algorithms

The recorded execution times (in seconds) compare the performance of two matrix multiplication algorithms and three matrix inversion algorithms across various input sizes. These execution times are presented in a table and visualised in a plot, with the y-axis on a logarithmic scale.

6.1 Matrix Multiplication

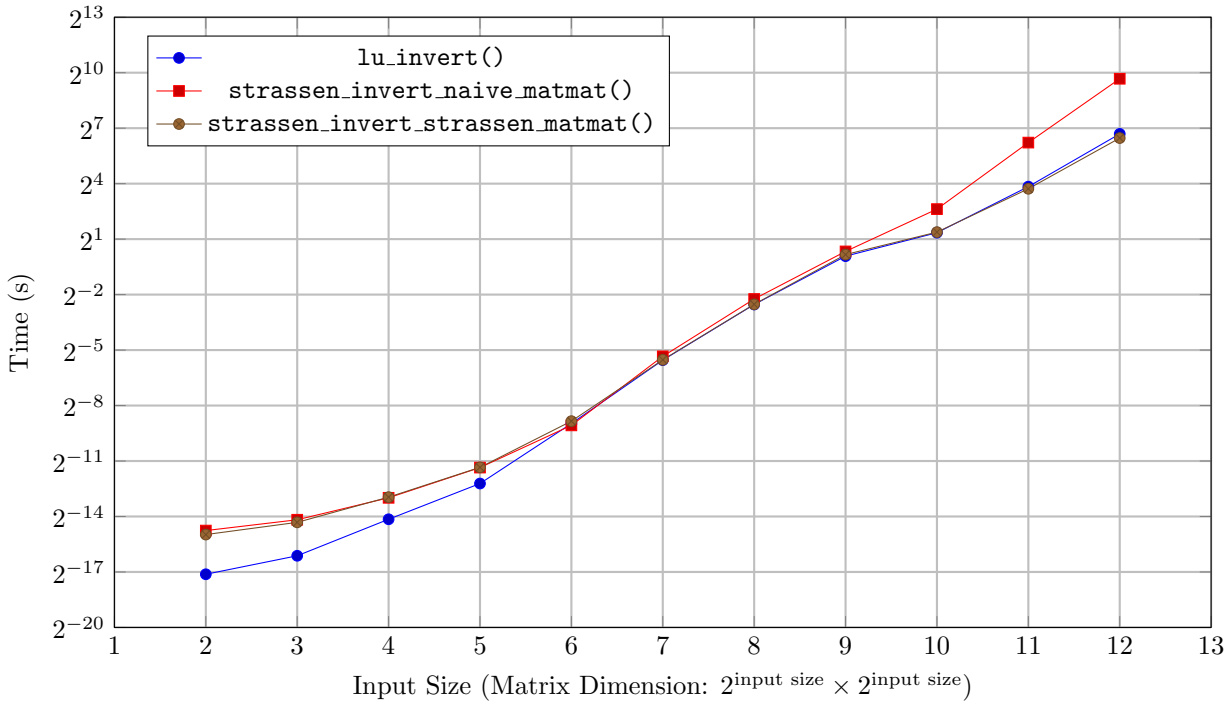


Input Size	`naive_matmat()`	`strassen_matmat()`
2	0.000010	0.000010
3	0.000010	0.000010
4	0.000040	0.000040
5	0.000430	0.000410
6	0.003160	0.002660
7	0.021410	0.016990
8	0.237320	0.179240
9	1.088840	1.054020
10	2.300790	2.726710
11	38.845700	12.486280
12	619.148860	83.786820

Table 1: Matrix Multiplication

For greater input sizes, the results show a noticeable performance gap between the two algorithms. At this point, Strassen no longer simply relies on the naive method, and its advantages become more evident. Both algorithms exhibit polynomial growth in execution time, with the naive algorithm failing to follow its expected cubic time complexity ($\mathcal{O}(n^3)$), where doubling the input size would have been expected to result in a eightfold increase in execution time. This may be due to a high likelihood of cache misses, as matrices are stored column-wise but require accessing contiguous columns in the second matrix during multiplication, leading to different cache lines. In contrast, `strassen_matmat()` demonstrates more efficient scaling, with a sub-cubic time complexity ($\mathcal{O}(n^{\log_2 7})$), where doubling the input size results in only a sevenfold increase in execution time, maintaining relatively stable execution times as input sizes increase.

6.2 Matrix Inversion



Input Size	`lu_invert()`	`s_i_naive_matmat()`	`s_i_strassen_matmat()`
2	0.000007	0.000036	0.000031
3	0.000014	0.000054	0.000049
4	0.000055	0.000123	0.000126
5	0.000210	0.000382	0.000385
6	0.001989	0.001858	0.002176
7	0.021437	0.024693	0.021671
8	0.172513	0.211797	0.173480
9	1.055680	1.260928	1.120102
10	2.543750	6.160911	2.600564
11	14.307355	74.543219	13.197856
12	103.317852	816.275145	88.330474

Table 2: Matrix Inversion

All algorithms exhibit polynomial growth in execution time. The `lu_invert()` algorithm follows the expected cubic time complexity ($\mathcal{O}(n^3)$), meaning that doubling the input size results in an eightfold increase in execution time. In contrast, `strassen_invert_strassen_matmat()` scales more efficiently, with a sub-cubic time complexity ($\mathcal{O}(n^{\log_2 7})$), where doubling the input size results in only a sevenfold increase in execution time. On the other hand, `strassen_invert_naive_matmat()` becomes increasingly dominated by the complexity of matrix multiplications. The time complexity advantage of the Strassen algorithm is particularly noticeable for larger input sizes, as the initial overhead is more significant for smaller ones.

7 Conclusion

Strassen's algorithm proves to be more efficient for both matrix multiplication and matrix inversion. Specifically, the Strassen inverse combined with Strassen matrix multiplication demonstrates superior performance compared to the inverse combined with naive matrix multiplication.

Matrix inversion is no harder than matrix multiplication [CLRS09]. Consequently, any improvement in matrix multiplication algorithms will directly enhance the efficiency of matrix inversion.

An analysis of the time complexity shows that the empirical runtime aligns closely with theoretical expectations. An exception to this is the naive matrix multiplication algorithm, where cache misses significantly increase empirical runtime compared to the theoretical predictions.

Potential improvements to the implementation include the definition of recursive access patterns and utility functions, which could reduce memory allocations. Additionally, padding could be optimised by avoiding complete memory copying and instead leveraging the original memory directly.

A System Information

Category	Details
System	5.10.16.3-microsoft-standard-WSL2
Distribution	Ubuntu 22.04.2 LTS
Architecture	x86_64
CPU	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx
Cores	4
Threads per Core	2
Cache (L1d)	128 KiB (4 instances)
Cache (L1i)	256 KiB (4 instances)
Cache (L2)	2 MiB (4 instances)
Cache (L3)	4 MiB (1 instance)
Memory	2.8 GiB (total)
Available Memory	2.5 GiB
Swap Memory	1.0 GiB

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[HSHvdG16] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. Strassen's algorithm reloaded. In *SC16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 690–701. IEEE Press, November 2016.

[JBD] Vincent Neiger Jérémy Berthomieu and Mohab Safey El Din. Numerical and symbolic algorithms modeling (model, mu4in901). *Sorbonne University lecture notes*.