Summer Intern at Polsys team, LIP6

High performance implementations of matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$

XU Audic

Internship carried out from July 2023 to August 2023

Internship tutor : Jérémy Berthomieu

Referent teacher : Jean-Lou Desbarbieux

Educational institution : Sorbonne Université - L3 Double licence Maths/Info

Internship host laboratory : LIP6 - 4 Pl. Jussieu, 75005 Paris

Abstract

During my second year of undergraduate studies(L2), I had the opportunity to explore the topic of "High performance implementation of matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$" under the guidance of M. Safey El Din, a researcher at LIP6. Now, for my summer internship, I have chosen to continue working on the same topic in collaboration with my supervisor M. Berthomieu, researcher at LIP6.

In this study, I will represent my entries as floating-point numbers and implement the modular operation technique introduced in the article "Modular SIMD arithmetic in MATHEMAGIX" [VLQ16]. This technique, only suitable for floating-point, allows efficient computations over finite fields. To improve performance, I will use parallelism with OpenMP and apply block product techniques using OpenBLAS.

During this internship, I aim to develop my knowledge in the field of high-performance computing(HPC). I'm excited to understand how HPC works and how it's used to solve complex problems. Working with experienced researchers at LIP6 will show me what it's like to be a researcher. I'll see how they work together and come up with new ideas. This experience will help me improve my skills and give me a better idea of how math and computers come together to solve problems.

Key words: High-performance computing, linear algebra, modular arithmetic, float precision, parallelism.

Contents

1 Introduction

Matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$ stands at the intersection of two fundamental pillars of mathematics: Linear Algebra and Arithmetic. This basic operation holds significant importance across various domains: Cryptography, AI, Imagery.

In practical, many algebraic problems can be modeled into linear problems, where the matrix multiplication is crucial. Moreover, problems over $\mathbb{Z}$, $\mathbb{Q}$, and even $\mathbb{Z}[X]$, can be solved thanks to the matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$.

2 Work environment

2.1 LIP6 and PolSys

LIP6 is the computer science laboratory of Sorbonne Université, ex-Université Paris 6. It has 4 research axes:

- AI and Data Science
- Architecture, Systems, and Network
- Safety, Security and reliability
- Theory and mathematics of computing

During my summer internship, I had the privilege to be a part of the PolSys team, which specializes in developing algebraic algorithms to solve mathematical problems, with a specific focus on polynomial systems. It is internationally recognized as one of the leading groups in the area of solving systems of polynomial equations and inequalities using exact methods.

2.2 Workspace

During my internship, I had the opportunity to work with talented and enthusiastic individuals from my office 338 and the office 325. I would like to thank Robin Kouba[1], Kevin Tran[2] and Marie Bonboire[3]. Interacting and working with these colleagues not only enriched my internship experience but also improved the quality of my work.

[1] Intern and pursuing a Ph.D. of geometry algebraic.
[2] Intern and pursuing a Ph.D. of computer algebra.
[3] Intern and pursuing her last year of undergraduate in mathematics and computer science.

3 Definition of the problem

3.1 Mathematical definition

Let $\mathbb{Z}/p\mathbb{Z} = \mathbb{F}_p = \{\overline{0}, ..., \overline{p-1}\}$. be a finite field. Let $A = (a_{ij}), B = (b_{ij}) \in \mathbb{F}_p^{n\times n}$ be two square matrices of size $n \times n$. We want to compute their product $A \times B = C = (c_{ij}) \in \mathbb{F}_p^{n\times n}$.

3.2 Computational setup and Data-types

The elements of the finite field $\mathbb{F}_p$ are represented in double, the 64 bit floating point data-type with a mantissa size of 53. To maintain accuracy before any modulo, we have only considered $p < 2^{26.5}$.

Theorem 3.1. Let $p < 2^{26.5}$ and $a, b \in \mathbb{F}_p$ be stored as double, then the computation $a \times b$ fits in double.

Proof.

$$a \times b < 2^{26.5} \times 2^{26.5} = 2^{53}$$

□

Note: Most of the benchmark was done using $p = 2^{26} - 5$ as our prime number. Despite, $p \approx 2^{26.5}$ also works.

To code our implementations, we used in **C** language, and stored our entries in a 1-dimensional array for its compatibility with the library OpenBLAS that we will implement later. The coefficient $a_{i,j}$ will be represented as $A[i \times n + j]$ instead of $A[i][j]$.

3.3 State of the art

Some of the modern techniques and approaches used includes:

- Strassen's algorithm, having a complexity of $(n^{log_2(7)})$ instead of $O(n^3)$.
- Parallel algorithms, using parallelism with multiple threads.
- GPU acceleration, computing with thousands of threads.
- Block algorithms, techniques that partition matrices into blocks and perform matrix multiplication on these smaller blocks.
- Cache memory, accessing high-speed memory near the CPU.

However, many libraries and software already have high performance implementations of matrix multiplication over finite field. They are:

- FLINT (Fast Library for Number Theory), **C** library.
- NTL (Number Theory Library), **C++** library.
- SageMath, a mathematical open source software.
- FFLAS-FFPACK, a collection of libraries.

4 My Contribution

4.1 Tools and conception used

Throughout this internship, several techniques and approaches were employed to enhance the performance of matrix multiplication over $\mathbb{F}_p$. To begin with, we optimized the cache memory by changing the order of loops of our matrix multiplication. Secondly, we used the modulo reduction introduced in the article [VLQ16]. This reduction method avoids the computation of slow quotients by approximating them quite accurately and efficiently. Then, we imported 2 libraries: OpenMP for parallel programming and OpenBLAS to perform block products. Finally, we combined techniques and approaches to obtain our final implementation of matrix multiplication over $\mathbb{F}_p$.

4.2 Realization

4.2.1 Naive Implementation

Below is the pseudo-code for the naive matrix product:

Algorithm 1 Naive mp

Require: $A, B, C \in \mathbb{F}_p^{n\times n}$.
Ensure: C is the zero matrix.

1: **for** $i = 0$ **to** $n-1$ **do**
2: **for** $j = 0$ **to** $n-1$ **do**
3: **for** $k = 0$ **to** $n-1$ **do**
4: $C[i][j] \leftarrow C[i][j] + ((A[i][k] \times B[k][j]) \mod p)$
5: **for** $i = 0$ **to** $n-1$ **do**
6: **for** $j = 0$ **to** $n-1$ **do**
7: $C[i][j] \leftarrow C[i][j] \mod p$

4.2.2 Cache Memory

Definition 4.1. Cache memory is a small and high-speed memory in the CPU. Its purpose is to store frequently accessed data and the neighboring data that is contiguous in memory.

Theorem 4.2. We can change the order of loops (line 1, 2, and 3) in **Algorithm 1**, and there are 6 possibilities of loops.

Proof. Instructions at line 4 and 5 do not require any hypothesis about i, j, k . As for the number of loops, we have $3! = 6$ possibilities, they are: IJK, IKJ, JIK, JKI, KIJ and KJI. □

A first improvement we can think of, is to exploit the cache memory playing with the order of loops. As previously defined, we better access our matrix entries horizontally. Using

the IJK order of loops leads to multiplying one row with one column for each coefficient. To increase rows manipulation, we can consider the KIJ and IKJ order of loops.

We implemented the 6 possibilities of order of loops. Below is the pseudo-code for the naive matrix product without any modulo operation:

Algorithm 2 Mp KIJ

Require: $A, B, C \in \mathbb{F}_p^{n \times n}$.
Ensure: C is the zero matrix.

```
for k = 0 to n − 1 do
    for i = 0 to n − 1 do
        for j = 0 to n − 1 do
            C[i][j] ← A[i][k] × B[k][j]
```

The graph below presents the performance comparison between the 6 implementations.

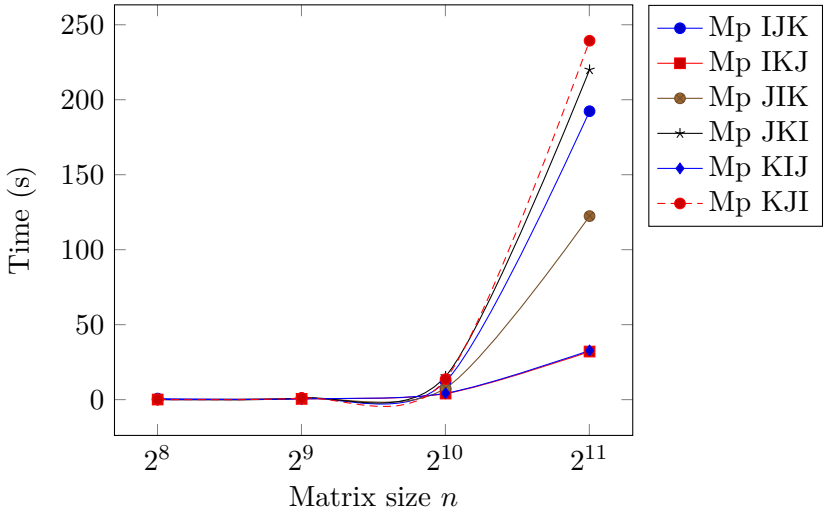

Fig. 1: Order of loops benchmark comparison

As expected, **KIJ** and **IKJ** performed the best, for the next implementations, we will keep **KIJ** as our default order of loops.

4.2.3 Reduction for double

To begin with, the native modulo operation in **C** language is slow, especially the computation of the quotient. Researchers J.Van Der Hoeven, G.Lecerf and G.Quintin published an article[1] where they performed an approximation of the quotient, leading to a more efficient modulo called reduction. This reduction requires to compute $u = 1/p$ stored as a double with $p < 2^{26.5}$.

Theorem 4.3. Let $a \in \mathbb{F}_p$ stored as a double and $q \in \mathbb{N}$ be the quotient of a divided by p. Let u be the computation of $\frac{1}{p}$ stored as a double.

Then, the value of $\lfloor a \times u \rfloor$ satisfies the following inequality:

$$q - 1 \leqslant \lfloor a \times u \rfloor \leqslant q + 1$$

Proof. cf Article [1]. □

Below is the implementation of this reduction called **Reduction 1**.

Algorithm 3 Reduction 1

Require: p, a prime number and $a, u \in \mathbb{F}_p$.
Ensure: $u = \frac{1}{p}$ using any rounding mode.

```
1: b ← a × u
2: c ← floor(b)
3: d ← a − c × p
4: if d ⩾ p then
5:     return d − p
6: if d < 0 then
7:     return d + p
8: return d
```

In the same article, they introduced a second reduction where they reduced the error range of the approximation. They computed $\widetilde{u} = \frac{1}{p}$ using the rounding mode towards infinity.

Theorem 4.4. Let $a \in \mathbb{F}_p$ stored as a double and $q \in \mathbb{N}$ be the quotient of a divided by p. Let $\widetilde{u}$ be the computation of $\frac{1}{p}$ using the rounding mode towards infinity.

Then, the value of $\lfloor a \times u \rfloor$ satisfies the following inequality:

$$q \leqslant \lfloor a \times u \rfloor \leqslant q + 1$$

Proof. We will only consider the first inequality which is the main idea of this theorem. We have:

$$\frac{1}{p} \leqslant \widetilde{u}$$

$$\frac{a}{p} \leqslant a \times \widetilde{u}$$

Since the floor function is increasing over $\mathbb{R}$, we have:

$$\left\lfloor \frac{a}{p} \right\rfloor \leqslant \lfloor a \times \widetilde{u} \rfloor$$

Hence, $q \leqslant \lfloor a \times \widetilde{u} \rfloor$.
Complete proof cf Article [1]. □

Below is the implementation with $\widetilde{u}$, called **Reduction 2**.

Algorithm 4 Reduction 2

Require: p, a prime number and $a, \widetilde{u} \in \mathbb{F}_p$.
Ensure: $\widetilde{u} = \frac{1}{p}$ using rounding mode towards infinity.

```
1: b ← a × ũ
2: c ← floor(b)
3: d ← a − c × p
4: if d < 0 then
5:     return d + p
6: return d
```

By thinking about parallelism which is the next subsection, we tried to make a **Reduction 3** function where we made the if statement to a parallelizable instruction.

Theorem 4.5. The code between line 4 and 6 in **Reduction 2** is equivalent to the instruction of **return** $d + (d < 0) \times p$.

Below is the pseudo-code of the implementation with the parallelizable instruction, called as **Reduction 3**.

Algorithm 5 Reduction 3

Require: p, a prime number and $a, \widetilde{u} \in \mathbb{F}_p$.
Ensure: $\widetilde{u} = \frac{1}{p}$ using rounding mode towards infinity.

```
1: b ← a × u
2: c ← floor(b)
3: d ← a − c × p
4: return d + (d < 0) × p
```

To conclude, we implemented matrix multiplication using **native C modulo**, **Reduction 1**, **Reduction 2** and **Reduction 3** and conducted rigorous performance tests. More precisely, we took **Naive mp** using the KIJ order, and replaced the modulo operator at line 4 and 5 and by each reduction. The benchmark protocol is available in the appendix.

The graph below presents the performance comparison for different modulo implementations with $p \approx 2^{26}$:

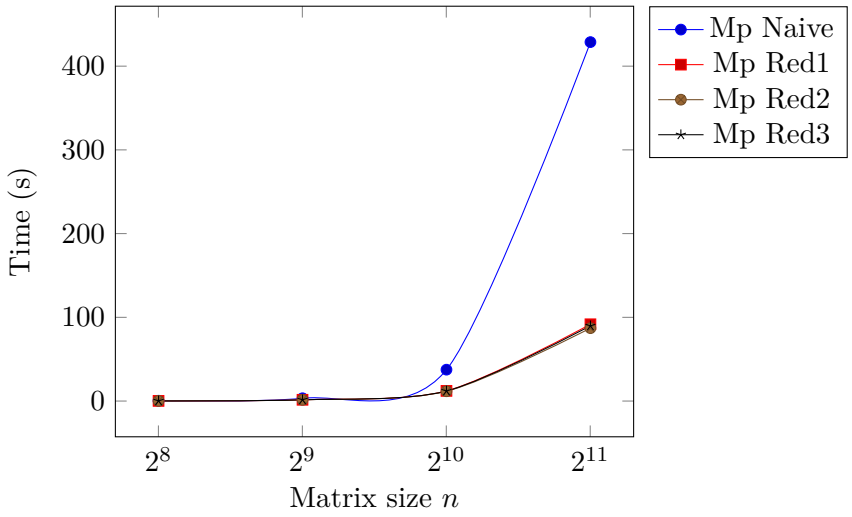

Fig. 2: Modulo techniques benchmark comparison

As expected, the implemented reductions are more efficient and we can see that the three reductions don't have significant performance differences between them. We decided to define **Reduction 3** as our default modulo.

4.2.4 Parallelism

Definition 4.6. Parallel computing is a type of computing architecture in which several processors simultaneously execute multiple, smaller calculations broken down from an overall larger, complex problem.

Since modern CPUs are multi-threaded, and the **Reduction 3** implementation is parallelizable, we imported OpenMP, an open source API for parallel programming.

We added parallelism approach to the previous implementations to obtain: **Naive mp KIJ: P** and the three matrix multiplication using reduction: **Mp Red1: P**, **Mp Red2: P** and **Mp Red3: P**. Below is the pseudo-code of **Naive Mp KIJ: P** and **Mp Red3: P**.

Algorithm 6 Naive mp KIJ: P

Require: $A, B, C \in \mathbb{F}_p^{n \times n}$.
Ensure: C is the zero matrix.

for $k = 0$ **to** $n - 1$ **do**
  #pragma omp parallel for
  for $i = 0$ **to** $n - 1$ **do**
    for $j = 0$ **to** $n - 1$ **do**
      $C[i][j] \leftarrow C[i][j] + ((A[i][k] \times B[k][j]) \mod p)$
#pragma omp parallel for
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $n - 1$ **do**
    $C[i][j] \leftarrow C[i][j] \mod p$

Algorithm 7 Mp Red3: P

Require: $A, B, C \in \mathbb{F}_p^{n \times n}$.
Ensure: C is the zero matrix and $\widetilde{u} = \frac{1}{p}$ using rounding monde towards infinity.

for $k = 0$ **to** $n - 1$ **do**
  #pragma omp parallel for
  for $i = 0$ **to** $n - 1$ **do**
    for $j = 0$ **to** $n - 1$ **do**
      $C[i][j] \leftarrow C[i][j] +$ **Reduction 3**$((A[i][k] \times B[k][j]), p, \widetilde{u})$
#pragma omp parallel for
for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $n - 1$ **do**
    $C[i][j] \leftarrow$ **Reduction 3**$(C[i][j], p, \widetilde{u})$

The graph below presents the performance comparison using parallelism with $p \approx 2^{26}$:

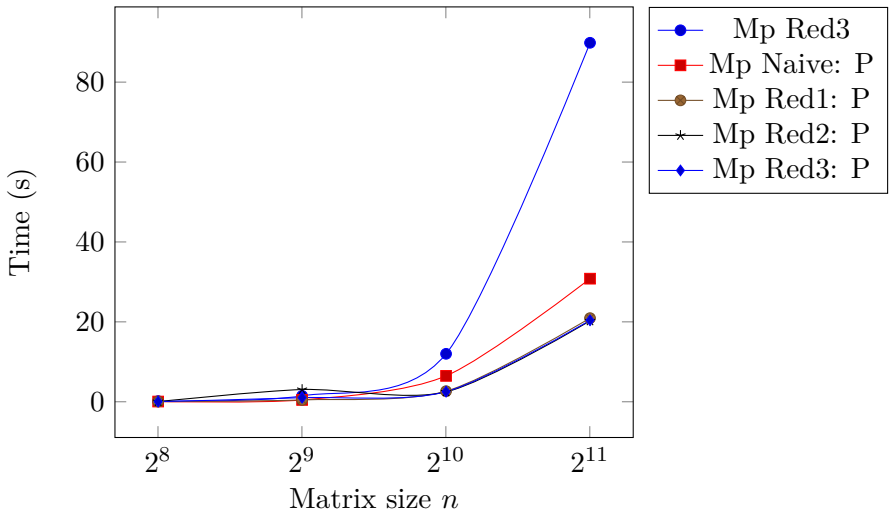

Fig. 3: Parallelized modulo benchmark comparison

From this Fig. 3, we can see an improvement of 200 percent when employing multiple threads. But contrary to what we expected for **Reduction 2** and **Reduction 3**, we don't observe any performance difference. To go further, we studied the frequency of entering the if case in **Reduction 2**. Unfortunately, for a matrix of size 1024×1024, the if statement is executed less than 10 times. So, it is not surprising that the performance between these implementations is similar.

4.2.5 Block algorithm technique

Theorem 4.7. Let A, B be two matrices of size $n \times n$. Then, **Naive Mp** has $n^3 + n^2$ modulo operations.

Because of the considerable amount of modulo operation, the goal is to minimize their occurrence as much as possible. To achieve this, we changed our way to perform matrix multiplication, using an alternative technique known as block product technique.

Definition 4.8. Let $A = \begin{bmatrix} A_1 & A_2 & \dots & A_N \end{bmatrix}$ and $B = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix}$ be two matrices of size $n \times n$ divided into submatrices of size $n \times \lambda$ and $\lambda \times n$ respectively. We have the following property:

$$A \times B = \sum_{i=1}^{N} A_i \times B_i \text{ with } N = \lfloor \frac{n}{\lambda} \rfloor.$$

The idea is to perform modulo operations after each $A_i \times B_i$ product, while avoiding modulo operations during the block product. To ensure exact results, we have to define a block size such that each coefficient of $A_i \times B_i$ block product remains in the representation limits of a double. Let λ represents the maximum permissible block size.

Theorem 4.9. Let $p < 2^k$ and $\lambda = \max(2^{53-2k}, n)$. Let $A \in \mathbb{F}_p^{n\times\lambda}$ and $B \in \mathbb{F}_p^{\lambda\times n}$ be two blocks. Then, each coefficient of the product $A \times B$ fits in double.

Proof. We denote $C = A \times B = (c_{ij})_{0\leqslant i,j\leqslant n-1}$.
For all $i, j \in \{0, \ldots, n-1\}$, we have:

$$\begin{aligned} c_{ij} &= \sum_{k=0}^{\lambda-1} a_{ik}b_{kj} \\ &< \sum_{k=0}^{\lambda-1} 2^k \times 2^k \\ &< \lambda \times 2^{53} \end{aligned}$$

Hence, for all $i, j \in \{0, \ldots, n-1\}$, $c_{i,j} < 2^{2k}$.

□

To perform block product technique, we imported OpenBLAS, a library that can perform block product operation efficiently. However, it's important to note that OpenBLAS does not support block product operations in a finite field, i.e., this library does not manage any modulo operations directly.

We introduced the block product approach into the matrix multiplication, below is the pseudo-code of **Mp Red3: NaiveB**, where 'NaiveB' denotes for naive blocks product, since we didn't rely on OpenBLAS.

Algorithm 8 Mp Red3: NaiveB

Require: $A, B, C \in \mathbb{F}_p^{n\times n}$.
Ensure: C is the zero matrix and $\widetilde{u} = \frac{1}{p}$ using the rounding mode towards infinity.

```
k ← 0
while k < n do
    k ← k + λ
    for kk = k to k + λ − 1 do
        for ii = 0 to n − 1 do
            for jj = 0 to n − 1 do
                C[ii][jj] ← C[ii][jj] + A[ii][kk] × B[kk][jj]
    for i = 0 to n − 1 do
        for j = 0 to n − 1 do
            C[i][j] ← Reduction 3(C[i][j], p, u)
```

Then, we implemented a matrix multiplication using the library OpenBLAS, below is the pseudo-code of **Mp Red3: B**:

Algorithm 9 Mp Red3: B

Require: $A, B, C \in \mathbb{F}_p^{n \times n}$.

Ensure: C is the zero matrix and $\widetilde{u} = \frac{1}{p}$ using the rounding mode towards infinity.

1: $k \leftarrow 0$
2: **while** $k < n$ **do**
3: $k \leftarrow k + \lambda$
4: **DGEMM**(Order=row, TransA=no, TransB=no, M=n, N=n, K=λ, alpha=1, PointerA=A+k, LDA=n, PointerB=B+k∗n, LDB=n, beta=1, PointerC=C, LDC=n)
5: **for** $i = 0$ **to** $n - 1$ **do**
6: **for** $j = 0$ **to** $n - 1$ **do**
7: $C[i][j] \leftarrow$ **Reduction 3**$(C[i][j], p, u)$

The graph below presents the performance comparison between **Mp Red3: NaiveB** and **Mp Red3: B** with $p \approx 2^{26}$:

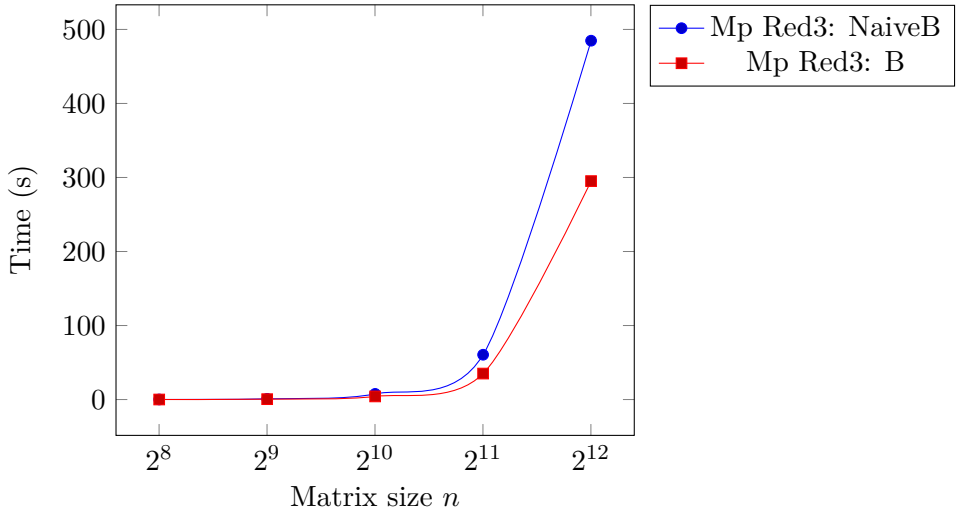

Fig. 4: Block product benchmark comparison

The results aligned our expectations. However what caught our attention was the observation that the number of threads used in OpenBLAS has a important impact on the implementation's speed. Surprisingly, using fewer threads leads to faster matrix multiplication. According to our experiments, we deduced a relationship between the optimal number of threads and the size of p, i.e., the size of the blocks. Although there is a potential optimization to be found in this area, due to the constraints of the internship schedule, we did not have enough time to study this part. We decided to use 1 thread to perform OpenBLAS's matrix multiplication.

4.3 Results and discussion

Below is a table to remind us what approaches and techniques we used during this internship, and how we named the implementations:

Implementation Name	Loops	Modulo	Parallelism: OpenMP	Block Prod: OpenBLAS
Naive Mp	IJK	Mod	No	No
Naive Mp KIJ	KIJ	Mod	No	No
Mp Red3	KIJ	Red 3	No	No
Mp Red3: P	KIJ	Red 3	Yes	No
Mp Red3: B	KIJ	Red 3	No	Yes
Mp Red3: P+B	KIJ	Red 3	Yes	Yes

Tab. 1: Summary of approaches and techniques for our final implementation

Before presenting the final results, as mentioned in the abstract, I studied the topic of "High performance implementations over $\mathbb{Z}/p\mathbb{Z}$", during my L2 year. Naturally, We wanted to compare the results of this internship with the results that I obtained last year. It's important to notice that, we conducted performance test within the same environment. Limiting to 1 thread only, using the same matrix entries and the same p.

For the purpose of this comparison, we employed **KIJ** as our order of loop, employed **Red3** reduction, used OpenBLAS with a single thread, called as **Float Mp**. Regarding the implementation from the previous year, I also employed Cache Memory optimization through matrix transpositions (equivalent to using 'KIJ' order), used Barrett's reduction (working for integers but slower than double's reduction), and finally employed a technique to reduce modulo operations, called decomposition into High and Low parts. This implementation will be referred to **Integer Mp**. The following graph presents the obtained results across various value of p and fixing the value of n as 4096:

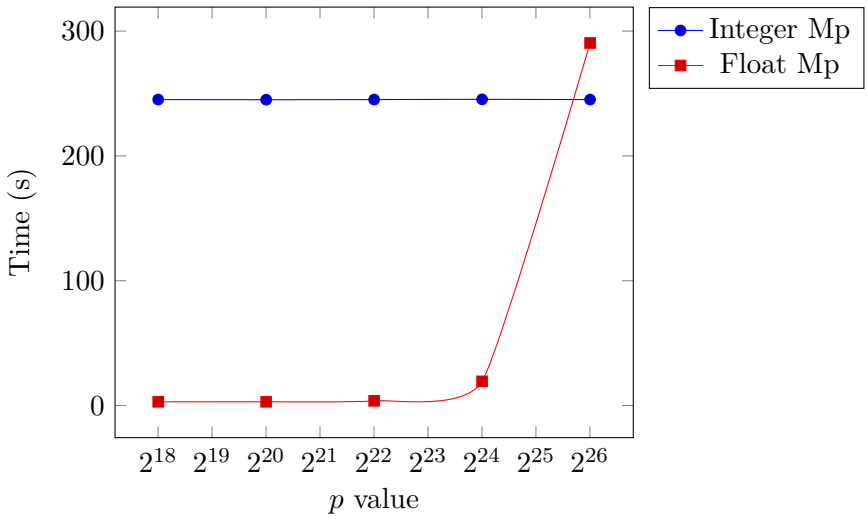

Fig. 5: Integer and float benchmark comparison with matrix size $n = 4096$

Below is the exact execution time of each implementation:

	$p \approx 2^{18}$	$p \approx 2^{20}$	$p \approx 2^{22}$	$p \approx 2^{24}$	$p \approx 2^{26}$
Integer Mp	245.12s	245.00s	245.12s	245.31s	245.10s
Float Mp	3.05s	3.04s	3.80s	10.38s	290.45s

Tab. 2: Benchmark between **Integer Mp** and **Float Mp** with matrix size $n = 4096$

By analyzing their asymptotic behavior, we observed that **Float Mp** follows a hyperbolic curve, primarily due to the variation in block size, resulting in varying block product efficiency. In contrast, **Integer Mp** has a constant execution time. Considering this, it is reasonable to conclude that the implementation obtained through this internship is more efficient, as long as $p < 2^{24}$. However, the implementation that I developed during my L2 year supports matrix multiplication for p up to 2^{31}, and can be considered as more stable and practical in case p is unknown.

Back to our final implementation with the potential of not being limited to 1 thread, we performed a benchmark for implementations present in Table 1. The graph below shows the performance comparison with $p \approx 2^{26}$:

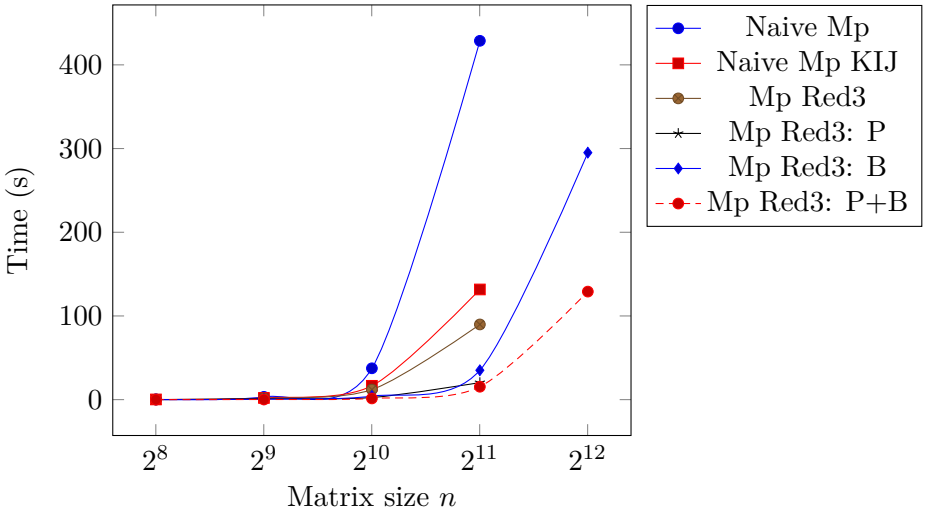

Fig. 6: Summary of benchmark comparisons

5 Conclusion and future directions

Throughout this internship, I gained insights into the life of a researcher, finding interesting and fascinating. I deepened my understanding of computer algebra and high performance computing. As a next step in my academic formation, I'm pursuing the European master's program of HPC at Sorbonne University. This internship has given an me excellent foundation to explore and contribute in these fields in the future.

References

[VLQ16] Joris Van Der Hoeven, Gregoire Lecerf, and Guillaume Quintin. "Modular SIMD arithmetic in Mathemagix". In: *ACM Transactions on Mathematical Software* (August 2016).

A Appendix

A.1 Benchmark Protocol

A.1.1 Library

For the purpose of benchmarking matrix multiplication over $\mathbb{Z}/p\mathbb{Z}$ implementations, I developed a library for matrices(square matrices). Below is the methods overview:

- **zero_matrix(size)** Generates a matrix filled with zeros.
- **fill_random_matrix(size, p)** Creates a matrix with random entries from $\{0, \ldots, p-1\}$.
- **convert_float_to_integer(mat, size)** Converts the entries of a floating-point matrix to integers.
- **delete_matrix(size)** Deallocates the memory occupied by the given matrix.
- **transpose_matrix(mat, size)** Computes the transpose of the given matrix.
- **print_matrix(mat, size)** Displays the coefficients of the matrix in output.
- **write_matrix_file(filename)** Writes matrix elements to a file with the given filename.
- **read_matrix_file(filename)** Reads matrix elements from a file with the given filename.
- **equals_matrix(mat1, mat2, size)** Compares if two matrices are equals.

A.1.2 Benchmark Procedure

Before starting the computation time measurement, a validation process was established to ensure the exactitude of the matrix multiplications results. For that, I used my library and employed Python as a powerful calculator, capable of handling large numbers with automatic memory allocation.

To measure the execution time of each implementation, I employed the **clock**() function from <time.h> to measure the CPU. However, I observed that when dealing with implementations using multiple threads, the **clock**() function did not give accurate results, it summed the time of each thread. To get around this, I used the **clock_gettime**() function from <time.h> and the **timespec** structure which resulted in accurate execution time results.