The Matrix Reloaded: Multiplication Strategies in FrodoKEM

Joppe W. Bos[1], Maximilian Ofner[2⋆], Joost Renes[1], Tobias Schneider[1], and Christine van Vredendaal[1]

[1] NXP Semiconductors
`firstname.lastname@nxp.com`
[2] TU Graz, Austria
`m.ofner@student.tugraz.at`

Abstract. Lattice-based schemes are promising candidates to replace the current public-key cryptographic infrastructure in wake of the looming threat of quantum computers. One of the Round 3 candidates of the ongoing NIST post-quantum standardization effort is FrodoKEM. It was designed to provide conservative security, which comes with the caveat that implementations are often bigger and slower compared to alternative schemes. In particular, the most time-consuming arithmetic operation of FrodoKEM is the multiplication of matrices with entries in $\mathbb{Z}_q$.
In this work, we investigate the performance of different matrix multiplication approaches in the specific setting of FrodoKEM. We consider both optimized "naïve" matrix multiplication with cubic complexity, as well as the Strassen multiplication algorithm which has a lower asymptotic run-time complexity. Our results show that for the proposed parameter sets of FrodoKEM we can improve over the state-of-the-art implementation with a row-wise blocking and packing approach, denoted as RWCF in the following. For the matrix multiplication in FrodoKEM, this results in a factor two speed-up. The impact of these improvements on the full decapsulation operation is up to 22 percent. We additionally show that for batching use-cases, where many inputs are processed at once, the Strassen approach can be the best choice from batch size 8 upwards. For a practically-relevant batch size of 128 inputs the observed speed-up is in the range of 5 to 11 percent over using the efficient RWCF approach and this speed-up grows with the batch size.

Keywords: Post-Quantum Cryptography · Matrix Multiplication · Software Implementation · Strassen.

1 Introduction

The security of nearly all our digital assets as well as our online activities relies on the hardness of the underlying cryptographic primitives. *Public-key cryptography*, most notably RSA [37] and Elliptic Curve Cryptography [27,30], is one of

⋆ This work was performed while this author was an internship student at NXP Semiconductors.

the fundamental components to establish a secure cryptographic infrastructure. With the steady progress in the development of quantum computers, the long-term security of this infrastructure, including encrypted information and digital signatures, is being threatened. When a full-scale quantum computer becomes available, all the currently standardized and widely-used public-key algorithms are vulnerable to polynomial-time attacks using a quantum computer [39,35].

As a reaction to this imminent threat on our currently deployed public-key infrastructure, the USA's National Institute of Standards and Technology (NIST) initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms in 2016 [32] where a new replacement standard is expected in 2024. These algorithms are known as *post-quantum* or *quantum-safe* algorithms. Arguably the most promising family of post-quantum secure cryptographic approaches are the *lattice-based* schemes. From its inception with Ajtai's seminal works [2,3], the field has grown to an active area of research (see e.g., [33] for a comprehensive overview).

Among the lattice-based family, the *learning with errors* (LWE) problem is a common foundation on which to construct practical and post-quantum secure schemes. It was first introduced by Regev in [36] and subsequently gained traction due to its hardness reduction proofs; the hardness of LWE (for certain parameterizations) can be reduced to the hardness of various worst-case lattice problems. To improve efficiency, multiple variants of the original LWE problem have been proposed. These use additional structures in the lattice to realize a faster and more compact version of LWE-based schemes. Notable examples are the Ring-LWE [29,34] and the Module-LWE [13,28] versions. While these variants indeed offer schemes with better performances, they are more removed from the original hardness proof of LWE.

In this paper, we focus on the NIST Round 3 candidate FrodoKEM [10,31]. It is derived from the base LWE problem and was designed to provide a practical post-quantum key exchange mechanism with conservative security. In particular, it is based on a carefully parameterized LWE problem, which is closely related to the conjectured-hard problems on *generic*, "algebraically unstructured" lattices. This makes it a very conservative and secure choice in practice.

The downside, of course, is that practical realizations of FrodoKEM are often bigger and slower compared to the algebraically structured alternatives, i.e., Kyber [11,38], NTRU [15], NTRU Prime [9,8] and Saber [17,18]. Still, the advance to the third round of the ongoing NIST standardization effort as well as being one of two post-quantum algorithms recommended by the German Federal Office for Information Security (BSI) as cryptographically suitable for long-term confidentiality [14] underline the practical relevance of FrodoKEM.

From a performance perspective, the most costly operations in FrodoKEM are the (pseudo-random) generation, multiplication and addition of large integer matrices. Hence, from an arithmetic point of view the main bottleneck and, therefore, focus of optimization is the implementation of the matrix multiplication. The main matrix used in these computations is a square integer matrix of dimension $n \in \{640, 976, 1344\}$ depending on the used parameter set of FrodoKEM. In the reference and optimized implementations of [31], this is achieved (if available) using the Advanced Vector Extensions (AVX) instructions on the `x64` platform. For matrix dimensions n, the implementations of [31] use a naïve matrix multiplication approach, i.e. of asymptotic complexity $\mathcal{O}(n^3)$. This is motivated by "street wisdom" that the asymptotically faster matrix multiplication algorithms only provide benefits for much larger values of n than used in FrodoKEM. Examples of such algorithms are the Strassen algorithm [41] ($\mathcal{O}(n^{\log_2(7)}) = \mathcal{O}(n^{2.807355})$), the Coppersmith–Winograd algorithm [16] ($\mathcal{O}(n^{2.375477})$), and the most recent improvements by Alman and Williams [6] ($\mathcal{O}(n^{2.3728596})$).

The concept of *batch cryptography* was first introduced by Fiat in [19]. He proposed to perform multiple encryptions or signature generations simultaneously in order to reduce the total complexity. This is achieved by batching the operations instead of performing them one-by-one (see Section 4.2 for more details and references). For certain use cases, which require the rapid processing of a large number of cryptographic operations, this approach can be very beneficial, e.g., one of the recent emerging technologies with such requirements is vehicular communication [12]. In the setting of FrodoKEM, batching could be used to decapsulate multiple encapsulated keys with the same private key, e.g., a server processing a multitude of client requests. In effect, this batch decapsulation would increase one dimension of the multiplied matrices in function of the processed queries.

Contributions. In this work, we investigate the validity of this "street wisdom". This is motivated and in line with the results from Huang, Smith, Henry, and van de Geijn [22] where they dispel some of the preconceived notions regarding the practicality of the Strassen matrix multiplication algorithm. They introduce various implementation strategies to make Strassen a viable alternative to and even outperform the naïve $\mathcal{O}(n^3)$ approach for much smaller dimensions than previously assumed. We apply the learnings from [22] to the cryptographic setting of FrodoKEM: matrix multiplication where one of the inputs is significantly smaller (dimension $\bar{n} \times n = 8 \times n$) compared to the other (dimension $n \times n$ matrix) with the added aspect that one matrix can be generated *on-the-fly*.

To this end, we first implement FrodoKEM with various approaches for matrix multiplication. In particular, we explore variations of naïve matrix multiplication and Strassen matrix multiplication. We show that using a row-wise blocking and packing approach, denoted as RWCF, combined with on-the-fly generation of the FrodoKEM matrix outperforms the current FrodoKEM matrix multiplication implementation by almost a factor two. When the RWCF approach is used in FrodoKEM decapsulation, we show an improvement of up to 22 percent.

Furthermore, we investigate the viability of Strassen for the batching use case. To this end, we benchmark the performance of FrodoKEM when computing batch operations. We show that for batch sizes as small as 8 (for FrodoKEM-1344), using the Strassen algorithm can provide better performances than the naïve multiplication and even the RWCF approach. For batch sizes 128 and upward, we show that we can expect improvements in the range 19 to 35 percent compared to the FrodoKEM matrix-multiplication method and of 5 to 11 percent

Table 1. The relevant FrodoKEM parameters for matrix multiplication for the various security levels.

Parameter set	NIST security level	q	$\bar{n}$	n
FrodoKEM-640	1	2^{15}	8	640
FrodoKEM-976	3	2^{16}	8	976
FrodoKEM-1344	5	2^{16}	8	1344

over the RWCF approach. As expected, the benefit of using Strassen becomes more significant as the batch size increases.

Outline. The remainder of this paper is structured as follows. In Section 2, we provide the necessary background on FrodoKEM and recursive matrix multiplication, in particular the Strassen algorithm. In Section 3, we outline the application of the different matrix multiplication approaches in the context of FrodoKEM. These are then benchmarked for FrodoKEM and batched FrodoKEM in Section 4, and the paper is concluded in Section 5.

2 Preliminaries

In this section we outline the basics of the FrodoKEM algorithm [10,31], with a focus on the generation of the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$. We also recall the Strassen [41] matrix multiplication algorithm.

Notation. We denote the ring of integers modulo q with $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Matrices are denoted with upper case boldface letters, e.g., $\mathbf{B} \in \mathbb{Z}_q^{m\times n}$, and its matrix element in the i-th row and j-th column as $\mathbf{B}_{i,j}$ (with $0 \leq i < m$ and $0 \leq j < n$).

2.1 The FrodoKEM Algorithm

FrodoKEM was derived from the Frodo key agreement protocol proposed in [10]. The security of Frodo reduces to the hardness of the standard Learning With Errors (LWE) problem with a short secret. In this section, we only recall the aspects of FrodoKEM relevant to our contribution. For further information, we refer the interested reader to the specification of FrodoKEM [31].

Table 1 contains the FrodoKEM parameters related to the matrix multiplication and their associated security levels. The NIST security levels 1, 3 and 5 correspond to the brute-force security of AES128, AES-192 and AES-256, respectively.

As can be seen from Table 1, the LWE integer modulus $q \leq 2^{16}$ is always a power of two in FrodoKEM. This was chosen for efficiency reasons: reduction modulo q is "for free" on modern computer architectures.

During the FrodoKEM key generation, secret and public keys are generated from an initial secret and public seeds. In particular, the *public* matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$

Algorithm 1 Frodo.Gen using AES128 (algorithm taken from [31]).

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$.

```
1: for ($i = 0$; $i < n$; $i \leftarrow i + 1$) do
2:     for ($j = 0$; $j < n$; $j \leftarrow j + 8$) do
3:         $\mathbf{b} \leftarrow \langle i\rangle \| \langle j\rangle \| 0 \cdots 0 \in \{0,1\}^{128}$ where $\langle i\rangle, \langle j\rangle \in \{0,1\}^{16}$
4:         $\langle c_{i,j}\rangle \| \langle c_{i,j+1}\rangle \| \cdots \| \langle c_{i,j+7}\rangle \leftarrow \mathrm{AES128}_{\mathsf{seed}_{\mathbf{A}}}(\mathbf{b})$ where each $\langle c_{i,k}\rangle \in \{0,1\}^{16}$
5:         for ($k = 0$; $k < 8$; $k \leftarrow k + 1$) do
6:             $\mathbf{A}_{i,j+k} \leftarrow c_{i,j+k} \bmod q$
7: return A
```

is created by calling FrodoKEM.Gen ($\mathsf{seed}_{\mathbf{A}}$) for the public seed $\mathsf{seed}_{\mathbf{A}}$. Given $\mathbf{A}$ and the *secret* matrix $\mathbf{S} \in \mathbb{Z}_q^{n\times \bar{n}}$, a second *public* matrix $\mathbf{B} \in \mathbb{Z}_q^{n\times \bar{n}}$ is computed as

$$\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}\,,$$

where $\mathbf{E}$ is randomly drawn from a (small) distribution χ. FrodoKEM security in this context relies on the hardness of recovering $\mathbf{S}$ from $\mathbf{B}$ and $\mathbf{A}$. The public key pk is then derived from $\mathbf{B}$ and $\mathsf{seed}_{\mathbf{A}}$, while the secret key sk further contains the secret seeds and matrices. Note that $\mathbf{A}$ is not part of any key and it is assumed to be always generated on-the-fly using $\mathsf{seed}_{\mathbf{A}}$.

Apart from error sampling and calls to symmetric primitives (i.e., AES128 or SHAKE128), the main operations in FrodoKEM are matrix operations. In the remainder of this section we focus on these operations.

To perform encryption FrodoPKE.Enc with respect to the matrix $\mathbf{A}$ one generates a secret matrix $\mathbf{S}' \in \mathbb{Z}_q^{\bar{n}\times n}$ and computes $\mathbf{B}' \in \mathbb{Z}_q^{\bar{n}\times n}$ as

$$\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'\,,$$

where $\mathbf{E}'$ is another matrix randomly drawn from a (small) distribution χ. Since encryption is a subroutine of both encapsulation and decapsulation, this matrix multiplication operation is also a critical component of FrodoKEM.Enc and FrodoKEM.Dec.

Generation of the Public Matrix A. To reduce the size of public keys and accelerate encryption, the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ could be set to a fixed value. However, the designers of FrodoKEM chose to assign the public matrix $\mathbf{A}$ dynamically and pseudorandomly generate it for every generated key. Following previous work in this area [5,10], using dynamic matrices $\mathbf{A}$ helps to avoid the possibility of backdoors and all-for-the-price-of-one attacks [1].

Let us recall how the matrix is constructed following the FrodoKEM specification [31]. The algorithm FrodoKEM.Gen takes as input the modulus q, a seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ and a dimension $n \in \mathbb{Z}$, and outputs a pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$. There are two options for instantiating FrodoKEM.Gen. The first method uses AES128; the second instead uses SHAKE128.

Algorithm 2 Frodo.Gen using SHAKE128 (algorithm taken from [31]).

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

```
1: for ($i = 0$; $i < n$; $i \leftarrow i + 1$) do
2:     $\langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow$ SHAKE128($\mathsf{seed}_{\mathbf{A}}, 16n, 2^8 + i$) where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$.
3:     for ($j = 0$; $j < n$; $j \leftarrow j + 1$) do
4:         $\mathbf{A}_{i,j} \leftarrow c_{i,j} \mod q$
5: return A
```

When using AES128, the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ is generated 8 elements at-a-time. For each row and each block of 8 elements (in different columns), the algorithm generates a 128-bit block of predefined input based on the location in the matrix. This input is encrypted using the $\mathsf{seed}_{\mathbf{A}}$ as the AES128 key. This process is outlined in Algorithm 1. More specifically, the input blocks to AES128 are $\langle i \rangle \| \langle j \rangle \| 0 \| \cdots \| 0 \in \{0,1\}^{128}$, where i, j are encoded as 16-bit integers (see Line 3). It then splits the 128-bit AES128 output block into eight 16-bit elements, which it interprets as non-negative integers $c_{i,j+k}$ for $k = 0, 1, \ldots, 7$ (see Line 4). Finally, it sets $\mathbf{A}_{i,j+k} = c_{i,j+k} \bmod q$ for all k. This modular reduction is "for free" by dropping the most significant bits whenever $q < 2^{16}$.

The second method uses SHAKE128 instead of AES128 to generate the rows of the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. This process is shown in Algorithm 2. In this case, each entire row is generated with a SHAKE128 call. Its input consists of $\mathsf{seed}_{\mathbf{A}}$ and a customization value $2^8 + i$ to produce a $16n$-bit output (see Line 2). The output is then split into 16-bit integers $c_{i,j} \in \{0,1\}^{16}$ (for $j = 0, 1, \ldots, n-1$), and used to set the corresponding matrix entries $\mathbf{A}_{i,j} = c_{i,j} \bmod q$ in Line 4. Note that the offset of 2^8 in the customization value is used for domain separation where for details we refer to the specification [31] of FrodoKEM.

2.2 The Strassen Algorithm

In this section we consider the application of the Strassen algorithm to the FrodoKEM multiplication $\mathbf{B}' = \mathbf{S}'\mathbf{A} + \mathbf{E}'$, where $\mathbf{B}', \mathbf{S}', \mathbf{E}' \in \mathbb{Z}_q^{\bar{n} \times n}$ and $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. The schoolbook approach of computing this sum would be to compute

$$\mathbf{B}'_{i,j} = \mathbf{E}'_{i,j} + \sum_{k=0}^{n-1} \mathbf{S}'_{i,k} \mathbf{A}_{k,j},$$

for each $i = 0, 1, \ldots, \bar{n} - 1$ and $j = 0, 1, \ldots, n - 1$. This requires $\bar{n}n^2$ multiplications of coefficients in $\mathbb{Z}_q$, and therefore is of complexity $\mathcal{O}(\bar{n}n^2)$. In the remainder we will refer to this specific multiplication method as the *straightforward* approach, while we will refer to $\mathcal{O}(\bar{n}n^2)$ methods in general as *naïve* approaches.

In 1969, Strassen introduced an algorithm for matrix multiplication [41] asymptotically faster compared to the straightforward approach. The Strassen algorithm works as follows. First the matrices $\mathbf{S}', \mathbf{A}$ and $\mathbf{E}'$ are split into four sub-matrices of equal size:

$$\mathbf{S}' = \begin{pmatrix} \mathbf{S}'_{00} & \mathbf{S}'_{01} \\ \mathbf{S}'_{10} & \mathbf{S}'_{11} \end{pmatrix}, \ \mathbf{A} = \begin{pmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} \\ \mathbf{A}_{10} & \mathbf{A}_{11} \end{pmatrix}, \ \mathbf{E}' = \begin{pmatrix} \mathbf{E}'_{00} & \mathbf{E}'_{01} \\ \mathbf{E}'_{10} & \mathbf{E}'_{11} \end{pmatrix},$$

where the sub-matrices of $\mathbf{S}'$ and $\mathbf{E}'$ are of dimension $\bar{n}/2 \times n/2$ and the sub-matrices of $\mathbf{A}$ of dimension $n/2 \times n/2$ each. The straightforward method would then be to compute

$$\mathbf{B}' = \begin{pmatrix} \mathbf{B}'_{00} & \mathbf{B}'_{01} \\ \mathbf{B}'_{10} & \mathbf{B}'_{11} \end{pmatrix},$$

where

$$\begin{aligned}
\mathbf{B}'_{00} &= \mathbf{E}'_{00} + \mathbf{S}'_{00} \cdot \mathbf{A}_{00} + \mathbf{S}'_{01} \cdot \mathbf{A}_{10}\,, \\
\mathbf{B}'_{01} &= \mathbf{E}'_{01} + \mathbf{S}'_{00} \cdot \mathbf{A}_{01} + \mathbf{S}'_{01} \cdot \mathbf{A}_{11}\,, \\
\mathbf{B}'_{10} &= \mathbf{E}'_{10} + \mathbf{S}'_{10} \cdot \mathbf{A}_{00} + \mathbf{S}'_{11} \cdot \mathbf{A}_{10}\,, \\
\mathbf{B}'_{11} &= \mathbf{E}'_{11} + \mathbf{S}'_{10} \cdot \mathbf{A}_{01} + \mathbf{S}'_{11} \cdot \mathbf{A}_{11}\,.
\end{aligned}$$

This split computation consists of eight products of dimension $\bar{n}/2 \times n/2$ sub-matrices with dimension $n/2 \times n/2$ sub-matrices, and does not decrease the overall number of multiplications compared to the straightforward approach. The idea by Strassen is to compute this instead as

$$\begin{aligned}
\mathbf{M}_0 &= (\mathbf{S}'_{00} + \mathbf{S}'_{11}) \cdot (\mathbf{A}_{00} + \mathbf{A}_{11})\,, & \mathbf{B}'_{00} &= \mathbf{E}'_{00} + \mathbf{M}_0 + \mathbf{M}_3 - \mathbf{M}_4 + \mathbf{M}_6\,, \\
\mathbf{M}_1 &= (\mathbf{S}'_{10} + \mathbf{S}'_{11}) \cdot \mathbf{A}_{00}\,, & \mathbf{B}'_{01} &= \mathbf{E}'_{01} + \mathbf{M}_2 + \mathbf{M}_4\,, \\
\mathbf{M}_2 &= \mathbf{S}'_{00} \cdot (\mathbf{A}_{01} - \mathbf{A}_{11}) & \mathbf{B}'_{10} &= \mathbf{E}'_{10} + \mathbf{M}_1 + \mathbf{M}_3\,, \\
\mathbf{M}_3 &= \mathbf{S}'_{11} \cdot (\mathbf{A}_{10} - \mathbf{A}_{00})\,, & \mathbf{B}'_{11} &= \mathbf{E}'_{11} + \mathbf{M}_0 - \mathbf{M}_1 + \mathbf{M}_2 + \mathbf{M}_5\,. \\
\mathbf{M}_4 &= (\mathbf{S}'_{00} + \mathbf{S}'_{01}) \cdot \mathbf{A}_{11}\,, \\
\mathbf{M}_5 &= (\mathbf{S}'_{10} - \mathbf{S}'_{00}) \cdot (\mathbf{A}_{00} + \mathbf{A}_{01})\,, \\
\mathbf{M}_6 &= (\mathbf{S}'_{01} - \mathbf{S}'_{11}) \cdot (\mathbf{A}_{10} + \mathbf{A}_{11})\,,
\end{aligned}$$

This requires only seven multiplications of dimension $\bar{n}/2 \times n/2$ sub-matrices with dimension $n/2 \times n/2$ sub-matrices, but has an increased number of additions and subtractions compared to the naïve method. Strassen's algorithm applies this splitting recursively, which asymptotically outperforms the straightforward block-by-block computation. For example, applying the recursion $\log_2 \bar{n}$ times leads to a complexity of $\mathcal{O}(\bar{n}^{\log_2 7 - 2} \cdot n^2)$, which is asymptotically better than the complexity of $\mathcal{O}(\bar{n}n^2)$ of naïve methods. The optimal number of recursion levels will depend on the various parameters and the platform on which the algorithm is implemented. In [22] it was shown that different strategies can reduce the overhead of Strassen and that the algorithm can show good results for smaller dimensions than previously known.

3 Matrix Multiplication Strategies for Cryptography

In this section, we present different strategies to realize efficient and practical implementations of matrix multiplication algorithms. These methods have been studied extensively in the literature before and are not new. The cryptographic application to FrodoKEM, however, which comes with the different setting of integer matrices where one of the operands is generated on-the-fly, has as far as we are aware not been considered in detail before. We use the techniques as proposed in the BLAS-like Library Instantiation Software (BLIS) framework [42], which is the infrastructure for instantiating Basic Linear Algebra Subprograms (BLAS) functionality. The core design idea is that virtually all BLAS operations (such as matrix-vector and matrix-matrix multiplications) can be expressed and optimized in terms of very simple kernels. Moreover, we use and describe the optimizing strategies as summarized and studied in [40].

3.1 Matrix Multiplication for FrodoKEM

In the setting of FrodoKEM we are particularly interested in the matrix product with accumulation. That is, we consider the operations $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ and $\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ where $\mathbf{S}, \mathbf{E} \in \mathbb{Z}_q^{n \times \bar{n}}$ and $\mathbf{S}', \mathbf{E}' \in \mathbb{Z}_q^{\bar{n} \times n}$ are sampled from a rounded continuous Gaussian distribution and $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ is generated pseudo-randomly from a seed $\mathtt{seed}_{\mathbf{A}}$ according to Algorithm 1 or Algorithm 2. For simplicity, we only focus on the generation of $\mathbf{A}$ with AES128 in this section. In the proposed parameter sets in [31], one uses $\overline{n} = 8$ and $n \in \{640, 976, 1344\}$ (see Table 1) and therefore the public matrix $\mathbf{A}$ is quite large: 800, 1860 and 3528 kilobytes, respectively.

We begin with a brief discussion on the multiplication $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$, which is the most straightforward. In the FrodoKEM submission this is performed with the naïve (schoolbook) matrix multiplication $\mathbf{B}_{i,j} = \sum_{k=0}^{n-1} \mathbf{A}_{i,k}\mathbf{S}_{k,j} + \mathbf{E}_{i,j}$. Note that this works particularly well with on-the-fly matrix generation: since each $\mathbf{B}_{i,j}$ only depends on the i-th row of $\mathbf{A}$, and since $\mathbf{A}$ is generated *row-wise* (see Section 2.1), one can generate a row of $\mathbf{A}$ and use it to generate all $\bar{n}$ elements in the same row of $\mathbf{B}$. This also sets itself up well for using 16-way SIMD 16-bit integer instructions (like AVX and AVX2 [23]), but hand-optimizing those results in only a one percent performance improvement due to the compiler being able to generate such optimized code very well [31, Section 3.2.1]. Hence, in this work we make no improvements to the multiplication $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$.

Instead, we consider the matrix operation $\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$. In this case, the naïve computation $\mathbf{B}'_{i,j} = \sum_{k=0}^{n-1} \mathbf{S}'_{i,k}\mathbf{A}_{k,j} + \mathbf{E}'_{i,j}$ relies on the j-th *column* of $\mathbf{A}$. This leads to a non-trivial problem for on-the-fly computation, as the matrix $\mathbf{A}$ is generated row by row. In the case of AES128 the situation is actually slightly simpler, as $\mathbf{A}$ is really only generated 8 row-elements at a time. However, we shall see that the choice of algorithm still has significant impact on performance. In the remainder of this section we compare various algorithms to compute $\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$.

3.2 The FrodoKEM Algorithm

The idea of FrodoKEM for computing $\mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ is simple to describe: since elements of a row of $\mathbf{A}$ are generated 8 columns at a time, the elements of $\mathbf{B}'$ are also generated 8 columns at a time. That is, for a fixed j one generates

$$\mathbf{A}_{i,j} \| \mathbf{A}_{i,j+1} \| \cdots \| \mathbf{A}_{i,j+7} \leftarrow \mathrm{AES128}_{\mathsf{seed}_\mathbf{A}}(\langle i \rangle \| \langle j \rangle \| 0 \cdots \| 0)$$

for all $i = 0, 1, \ldots, n-1$ according to Algorithm 1. These elements can then be used to compute $\mathbf{B}'_{k,j}, \ldots, \mathbf{B}'_{k,j+7}$ for $k = 0, 1, \ldots, \bar{n} - 1$.

The most straightforward way to implement this, which is done by FrodoKEM, is to store the input to AES128 as a sequence of n blocks of 128-bit each

$$\langle 0 \rangle \| \langle j \rangle \| 0 \| \cdots \| 0 \| \langle 1 \rangle \| \langle j \rangle \| 0 \| \cdots \| 0 \| \cdots \| \langle n-1 \rangle \| \langle j \rangle \| 0 \| \cdots \| 0 \,,$$

to which AES128 can be applied independently. As a result, the elements of the 8 columns of $\mathbf{A}$ are stored sequentially as

$$\mathbf{A}_{0,j} \| \cdots \| \mathbf{A}_{0,j+7} \| \mathbf{A}_{1,j} \| \cdots \| \mathbf{A}_{1,j+7} \| \cdots \| \mathbf{A}_{n-1,j} \| \cdots \| \mathbf{A}_{n-1,j+7} \,.$$

However, to compute $\mathbf{B}'_{0,j}$ one would need to access $\mathbf{A}_{0,j}, \mathbf{A}_{1,j}, \ldots, \mathbf{A}_{n-1,j}$ which are *not* stored sequentially in memory. To solve this, FrodoKEM explicitly converts the representation to

$$\mathbf{A}_{0,j} \| \cdots \| \mathbf{A}_{n-1,j} \| \mathbf{A}_{0,j+1} \| \cdots \| \mathbf{A}_{n-1,j+1} \| \cdots \| \mathbf{A}_{0,j+7} \| \cdots \| \mathbf{A}_{n-1,j+7} \,,$$

which is essentially a *transpose* of the columns of $\mathbf{A}$. We observe that this memory re-organization does have a significant impact on the efficiency of the algorithm (cf. Table 2 in Section 4). For completeness, we summarize the matrix multiplication algorithm of FrodoKEM in Algorithm 3, where the transposition is performed in Lines 12 to 14. The authors of FrodoKEM made an efficient implementation of Algorithm 3 available where the multiplications and additions of the matrix elements are computed using the 256-bit Advanced Vector Extensions (AVX) 16-way SIMD 16-bit integer instructions [4].

3.3 The RWCF approach: Row-wise Cache-Friendly Multiplication

In this paper we look at an alternative approach to implement the same straightforward matrix multiplication algorithm with asymptotic run-time $\mathcal{O}(n^2\bar{n})$. We follow the blocking and packing approach as outlined in [20,42] which does not seem to have been considered for the FrodoKEM submission. Note that the multiplication with this complexity still falls under the naïve matrix multiplication methods. The idea is to avoid the expensive transposition in memory required by the FrodoKEM algorithm, which leads to an improvement in performance.

For this purpose, the elements of $\mathbf{A}$ are generated *row-wise* as opposed to column-wise. This is done 8 rows at a time in our benchmarked implementation, as this led to the best performance. However, doing fewer or more is possible as there is no dependency between different rows (as opposed to columns). For

Algorithm 3 Matrix multiplication as implemented in the official FrodoKEM submission when using AES128. The temporary memory buffers used are $\mathbf{A}^{\text{cols}}, \mathbf{T}$ and $\mathbf{AT}^{\text{cols}}$ of $8n$ elements of $\mathbb{Z}_q$ each.

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ and matrices $\mathbf{S}', \mathbf{E}' \in \mathbb{Z}_q^{\bar{n} \times n}$.
Output: $\mathbb{Z}_q^{\bar{n} \times n} \ni \text{out} = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$.

```
for $i \leftarrow 0$; $i < \bar{n}$; $i \leftarrow i+1$ do
    for $j \leftarrow 0$; $j < n$; $j \leftarrow j+1$ do
        $\text{out}_{i,j} \leftarrow \mathbf{E}'_{i,j}$
Set $\mathbf{T}$ to all zeros
$\text{aes}_k \leftarrow$ AES128_load_key_schedule($\mathsf{seed}_{\mathbf{A}}$)
for $i \leftarrow 0$; $i < n$; $i \leftarrow i+1$ do
    $\mathbf{T}[8i] \leftarrow i$
for $k \leftarrow 0$; $k < n$; $k \leftarrow k+8$ do
    for $i \leftarrow 0$; $i < n$; $i \leftarrow i+1$ do
        $\mathbf{T}[8i+1] \leftarrow k$
    $\mathbf{A}^{\text{cols}} \leftarrow \text{AES128\_ECB}_{\text{aes}_k}(\mathbf{T})$
    for $i \leftarrow 0$; $i < n$; $i \leftarrow i+1$ do
        for $j \leftarrow 0$; $j < 8$; $j \leftarrow j+1$ do
            $\mathbf{AT}^{\text{cols}}[j \cdot n + i] \leftarrow \mathbf{A}^{\text{cols}}[8i+j]$          // Transpose
    for $i \leftarrow 0$; $i < \bar{n}$; $i \leftarrow i+1$ do
        for $\ell \leftarrow 0$; $\ell < 8$; $\ell \leftarrow \ell+1$ do
            $\text{sum} \leftarrow 0$
            for $j \leftarrow 0$; $j < n$; $j \leftarrow j+1$ do
                $\text{sum} \leftarrow \text{sum} + \mathbf{S}'_{i,j} \cdot \mathbf{AT}^{\text{cols}}[\ell \cdot n + j]$     // Access AT sequentially
            $\text{out}_{i,k+\ell} \leftarrow \text{out}_{i,k+\ell} + \text{sum}$
```

simplicity, we describe the approach for a single row, as using more rows can be deduced easily by doing them in parallel. We provide the full description for 8 rows in Algorithm 4.

For a fixed row k, the input to AES128 is generated (sequentially in memory) as

$$\langle k \rangle \| \langle 0 \rangle \| 0 \| \cdots \| 0 \| \langle k \rangle \| \langle 8 \rangle \| 0 \| \cdots \| 0 \| \cdots \| \langle k \rangle \| \langle n-8 \rangle \| 0 \| \cdots \| 0 \,,$$

to which we apply AES128 to obtain

$$\mathbf{A}_{k,0} \| \mathbf{A}_{k,1} \| \cdots \| \mathbf{A}_{k,n-1} \,.$$

We then initialize $\mathbf{B}'^{(-1)} = \mathbf{E}'$ and iteratively accumulate $\mathbf{B}'^{(k)}$ as

$$\mathbf{B}'^{(k)}_{i,j} = \mathbf{B}'^{(k-1)}_{i,j} + \mathbf{S}'_{i,k} \mathbf{A}_{k,j} \,, \tag{1}$$

for all $i = 0, 1 \ldots, \bar{n}-1$ and $j = 0, 1 \ldots, n-1$. One sees that

$$\mathbf{B}'_{i,j} = \mathbf{B}'^{(n-1)}_{i,j} = \mathbf{E}'_{i,j} + \sum_{k=0}^{n-1} \mathbf{S}'_{i,k} \mathbf{A}_{k,j} \,,$$

proving correctness of the algorithm. Moreover, the elements of $\mathbf{A}$ are accessed in the same order in which they are generated, making this algorithm very suitable to on-the-fly generation.

Algorithm 4 Matrix multiplication in FrodoKEM with row-wise AES128 generation. The temporary memory buffers used are $\mathbf{A}^{\text{rows}}$ and $\mathbf{T}$ of $8n$ elements of $\mathbb{Z}_q$ each.

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ and matrices $\mathbf{S}', \mathbf{E}' \in \mathbb{Z}_q^{\bar{n}\times n}$.
Output: $\mathbb{Z}_q^{\bar{n}\times n} \ni \mathbf{B}' = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$.

```
1:  for $i \leftarrow 0; i < \bar{n}; i \leftarrow i+1$ do
2:      for $j \leftarrow 0; j < n; j \leftarrow j+1$ do
3:          $\mathbf{B}'_{i,j} \leftarrow \mathbf{E}'_{i,j}$
4:  Set $\mathbf{T}$ to all zeros
5:  $\text{aes}_k \leftarrow$ AES128_load_key_schedule($\mathsf{seed}_{\mathbf{A}}$)
6:  for $i \leftarrow 0; i < n; i \leftarrow i+8$ do
7:      for $j \leftarrow 0; j < 8; j \leftarrow j+1$ do
8:          $\mathbf{T}[j \cdot n + i + 1] \leftarrow i$
9:  for $i \leftarrow 0; i < n; i \leftarrow i+8$ do
10:     for $j \leftarrow 0; j < 8; j \leftarrow j+1$ do
11:         for $k \leftarrow 0; k < n; k \leftarrow k+8$ do
12:             $\mathbf{T}[j \cdot n + k] \leftarrow i + j$
13:     $\mathbf{A}^{\text{rows}} \leftarrow \text{AES128\_ECB}_{\text{aes}_k}(\mathbf{T})$
14:     for $j \leftarrow 0; j < \bar{n}; j \leftarrow j+1$ do
15:         for $\ell \leftarrow 0; \ell < n; \ell \leftarrow \ell+1$ do
16:             sum $= \mathbf{B}'_{j,\ell}$
17:             for $k \leftarrow 0; k < 8; k \leftarrow k+1$ do
18:                 sum = sum + $\mathbf{S}'_{j,i+k} \cdot \mathbf{A}^{\text{rows}}[k \cdot n + \ell]$        // 16 in parallel (AVX2)
19:             $\mathbf{B}'_{j,\ell} \leftarrow$ sum
```

This approach can be combined very efficiently with the available SIMD extensions. Specifically, one can broadcast the (16-bit) value $\mathbf{S}'_{i,k}$ to the 16 SIMD slots. This broadcast is done using the AVX instruction `_mm256_set1_epi16`$(\cdot)$ which puts the 16-bit integer a in all 16 slots of the returned 256-bit vector register. These values can be multiplied with 16 matrix elements $\mathbf{A}_{k,j} \| \cdots \| \mathbf{A}_{k,j+15}$ using a single instruction: `_mm256_mullo_epi16`$(\cdot,\cdot)$. This computes the products $\mathbf{S}'_{i,k}\mathbf{A}_{k,j}, \ldots, \mathbf{S}'_{i,k}\mathbf{A}_{k,j+15}$, for the 16-bit integers $\mathbf{S}'_{i,k}$ and $\mathbf{A}_{k,j}, \ldots, \mathbf{A}_{k,j+15}$, and has the additional advantage that the obtained result is automatically reduced modulo q (or $2q$ when $q = 2^{15}$ is used). This can be applied in Line 18 of Algorithm 4. Note that $16 \mid n$ for all parameter sets of FrodoKEM, so generating 16 row elements of $\mathbf{A}$ at a time is not a problem.

It should be clear that the accumulation step in Equation (1) can be computed for multiple rows at the same time by generating those rows simultaneously for various k. Although the number of multiplications and additions does not change in that case, it can be beneficial for the overall run-time by reducing the overall loads and stores of the $\mathbf{B}'_{i,j}$. This is especially true when loads and stores are performed to and from AVX registers. For example, in Line 16 and 19 of Algorithm 4 there is only a single load and store of $\mathbf{B}'_{j,\ell}$ for every 8 accumulations.

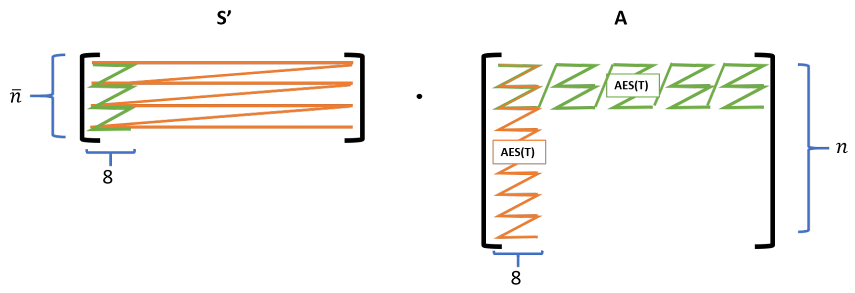


Fig. 1. Graphical representation of processing the elements. In orange the FrodoKEM approach, which for cache-friendly access requires transposing the blocks of columns from **A**, before multiplying with the rows of **S**. In green the RWCF approach, which does not require a transpose.

Note that for AES128-based version it is not actually necessary to generate a whole row of **A**: as we apply AES128 to 8 elements at a time, we can generate exactly those 8 elements (in the same row) on-the-fly (though 16 would be preferable for compatibility with AVX instructions). In that case we could consider another extreme version of the above algorithm where we process n rows simultaneously, generating 16 columns on-the-fly and multiplying and accumulating. This would reduce essentially to a column-based approach again, though the order of multiplications is different from FrodoKEM and a tranposition in memory is not necessary. However, since this algorithm is not compatible with SHAKE, which does generate whole rows from a single SHAKE call, we do not pursue this further here.

To illustrate the high-level difference in the order of accessing **A**, we present a simplified representation in Figure 1. In orange, we see the first columns of **A** are processed, which require an additional memory transposition. In green the row-wise method is shown, which needs no explicit memory transformations.

3.4 FrodoKEM Multiplication using Strassen

The last multiplication approach we discuss in the context of FrodoKEM is Strassen, which was already introduced in Section 2.2. In [22] it was shown that Strassen can also be implemented in a cache-friendly manner. The method presented there can be straightforwardly combined with the on-the-fly generation of **A**.

Recall that the AVX2 SIMD instructions allow us to process 16 16-bit elements at the same time, assuming that $16 \mid n$. To apply the same instructions for Strassen to the submatrices with only $n/2$ rows, we would require that $16 \mid (n/2)$. This is true for $n = 640, 1344$ but does not hold for $n = 976$. This is easily solved by padding **A** with zero columns which has a minor effect on the performance compared to the other parameter sets.

In the following, we only consider one-level Strassen and analyze its performance. That is, we reduce the matrix multiplication to 7 multiplications of half-size row and column dimensions that we perform with RWCF. More levels of recursion can of course be considered, but it is not a priori clear that Strassen will outperform other algorithms for the dimensions of FrodoKEM (even for a single recursion level), as its improvements are only guaranteed asymptotically. As we will see in Section 4, it outperforms the current FrodoKEM implementation but does not improve over RWCF itself. Nevertheless, even if Strassen does not scale fast enough to be relevant for single FrodoKEM, it can still be useful to explore its application for *batching*, as we show in Section 4.2.

4 Implementation and Benchmark Results

In this section we discuss the comparative performance for the different approaches of implementing the FrodoKEM matrix multiplication. The performance results have been obtained when running on a single core of the 12-core AMD Ryzen 9 3900XT running at a base clock of 3.8GHz. We consider both the setting where a single key exchange or encryption is performed, as well as their batched analogues where multiple keys are handled in parallel.

4.1 Performance Results

The performance measurements for all three FrodoKEM parameter sets are summarized in Table 2. This shows the performance in 10^3 cycles of the individual matrix multiplication routines $\mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ (`frodo_mul_add_as_plus_e`) and $\mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ (`frodo_mul_add_sa_plus_e`). These routines consist of two computationally significant steps: generation of the matrix elements of $\mathbf{A}$ using AES128 and multiplying the resulting matrix with $\mathbf{S}$ or $\mathbf{S}'$. Although a fresh $\mathbf{A}$ is generated for each IND-CPA encryption or key exchange, in a KEM setting where a static key pair is used one can pre-compute $\mathbf{A}$ and store for encapsulation and decapsulation. Therefore we distinguish two separate cases: excluding (labeled "pre") and including the generation time of the matrix elements from $\mathbf{A}$ using AES128. Note that the algorithms described in Section 3 only impact the matrix *multiplication* step and not the *generation*, so have relatively more impact when $\mathbf{A}$ is not freshly generated. For completeness, we also include the total cost of key generation, encapsulation and decapsulation, which generate $\mathbf{A}$ on-the-fly to align with the reference implementation. Again, the impact on encapsulation and decapsulation is greater by storing $\mathbf{A}$ in advance.

Firstly, we highlight an interesting observation about the reference implementation (using AVX instructions) of FrodoKEM (the "x64" column in Table 2). When comparing the two matrix multiplication routines, we see that computing $\mathbf{A} \cdot \mathbf{S} + \mathbf{E}$ is up to 1.4 times faster than $\mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ if the generation of $\mathbf{A}$ is included, and up to 1.9 times faster if $\mathbf{A}$ is pre-generated. The latter speed-up is almost fully determined by the matrix multiplication, but is surprisingly large since the dimensions of the multiplications are exactly the same (though transposed). Using the RWCF algorithm from Section 3.3 for $\mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ leads to a speed-up of up to 1.4 times including generation of $\mathbf{A}$, or of up to 2.0 times excluding it, when compared to the reference implementation. Indeed, the RWCF approach reduces the cost of $\mathbf{S}' \cdot \mathbf{A} + \mathbf{E}'$ so that it is essentially equal to computing $\mathbf{A} \cdot \mathbf{S} + \mathbf{E}$, which should be expected for multiplications of equal dimensions. Overall, employing the RWCF approach leads to an up to 22 percent improvement in encapsulation or decapsulation when $\mathbf{A}$ is generated on-the-fly, while not affecting key generation since it only computes $\mathbf{A} \cdot \mathbf{S} + \mathbf{E}$.

Interestingly, the Strassen implementation also outperforms the x64 implementation. This approach uses a single level of Strassen and then reverts to the RWCF approach for multiplying the smaller sub-matrices. This explains why Strassen outperforms x64 and not the RWCF approach. For the best overall performance one should use the RWCF approach. We expect that these results carry over to other approaches, compared to AES128, to generate the matrix elements. One such example is when using SHAKE128 as outlined in Section 2.1.

4.2 Batching

Let us consider the setting of batch cryptography [19]. The main idea is to reduce the computational burden of an entity which receives multiple (i.e., a batch of) cryptographic operations. It might be possible to process this batch of computation and take advantage of some arithmetic or algorithmic advantages that increase the latency (compared to a single request) but also increase the overall throughput (the number of cryptographic operations per second) to ensure an overall increase of computation on this batch processing system. Many of such approaches have been proposed such as batch verification of RSA signatures [21], ECDSA batch signature verification [25,26,24] and batch Diffie-Hellman key agreement [7].

In Frodo, the seed $\mathsf{seed}_{\mathbf{A}}$ used to generate the large public matrix $\mathbf{A}$ (on-the-fly) is part of the *public-key*. This means that when Frodo is used as a public-key encryption scheme multiple devices or clients can encrypt messages to be sent to the same server which can then perform a *batch decryption* on all these ciphertexts which use the same matrix $\mathbf{A}$. Along similar lines multiple clients can start the key encapsulation mechanism (using FrodoKEM) with the same clients using the same $\mathsf{seed}_{\mathbf{A}}$ and corresponding matrix $\mathbf{A}$. This allows for *batch decapsulation* on the server. This batching technique enables the server to use the same public matrix $\mathbf{A}$ for multiple requests and increase the dimension in the matrix multiplication $\mathbf{S} \cdot \mathbf{A}$ by considering multiple matrices $\mathbf{S}$ at once.

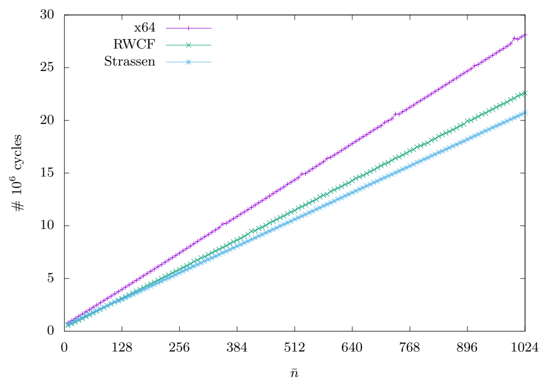
We investigate when (and if) the asymptotic performance gain of the Strassen algorithm becomes visible in such a batch decryption or batch decapsulation approach. Performance results when batching up to 128 computations (up to $\bar{n} = 8 \cdot 128 = 1024$) are shown in Figure 2 for the three parameters sets proposed in FrodoKEM. As expected eventually Strassen will outperform the FrodoKEM approach in all settings. We see an improvement of 26, 16 and 16 percent for FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344, respectively.

Table 2. Performance numbers of the matrix multiplication methods with and without ("pre") generation of the elements using AES128. In parentheses the relative performance against the reference implementation "x64". The numbers are reported in 10^3 cycles and an average over 1000 runs.

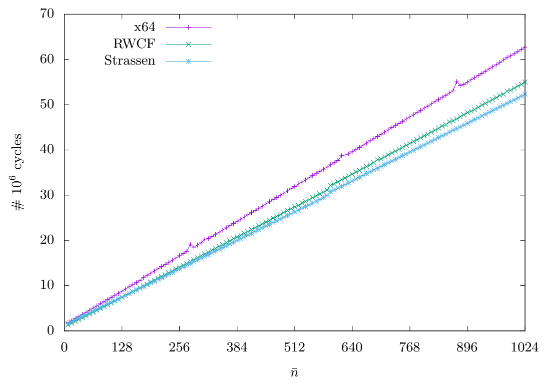
Function	x64	Strassen	RWCF
FrodoKEM-640			
`frodo_mul_add_as_plus_e` *(pre)*	208	212 (1.02)	212 (1.02)
`frodo_mul_add_sa_plus_e` *(pre)*	396	282 (0.71)	202 (0.51)
`frodo_mul_add_as_plus_e`	473	477 (1.01)	477 (1.01)
`frodo_mul_add_sa_plus_e`	661	547 (0.83)	467 (0.70)
`crypto_kem_keypair`	902	902 (1.00)	903 (1.00)
`crypto_kem_enc`	1 275	1 174 (0.92)	1 068 (0.84)
`crypto_kem_dec`	1 232	1 121 (0.91)	1 025 (0.83)
FrodoKEM-976			
`frodo_mul_add_as_plus_e` *(pre)*	507	508 (1.00)	501 (0.99)
`frodo_mul_add_sa_plus_e` *(pre)*	931	759 (0.82)	493 (0.53)
`frodo_mul_add_as_plus_e`	1 095	1 096 (1.00)	1 089 (0.99)
`frodo_mul_add_sa_plus_e`	1 519	1 347 (0.89)	1 081 (0.71)
`crypto_kem_keypair`	1 718	1 727 (1.01)	1 712 (1.00)
`crypto_kem_enc`	2 398	2 246 (0.94)	1 955 (0.82)
`crypto_kem_dec`	2 310	2 141 (0.93)	1 850 (0.80)
FrodoKEM-1344			
`frodo_mul_add_as_plus_e` *(pre)*	1 060	1 031 (0.97)	1 024 (0.97)
`frodo_mul_add_sa_plus_e` *(pre)*	1 888	1 412 (0.75)	1 012 (0.54)
`frodo_mul_add_as_plus_e`	2 140	2 111 (0.99)	2 104 (0.98)
`frodo_mul_add_sa_plus_e`	2 968	2 492 (0.84)	2 092 (0.70)
`crypto_kem_keypair`	3 070	3 023 (0.98)	3 017 (0.98)
`crypto_kem_enc`	4 279	3 777 (0.88)	3 363 (0.79)
`crypto_kem_dec`	4 130	3 634 (0.88)	3 221 (0.78)

This shows that for the batching use case, performing one-level Strassen becomes a viable option for the parameter sizes of FrodoKEM. Strassen also eventually outperforms the RWCF approach in all settings. The cross-over point is at $\overline{n}$ equal to 120, 152, 64, for FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344, respectively.

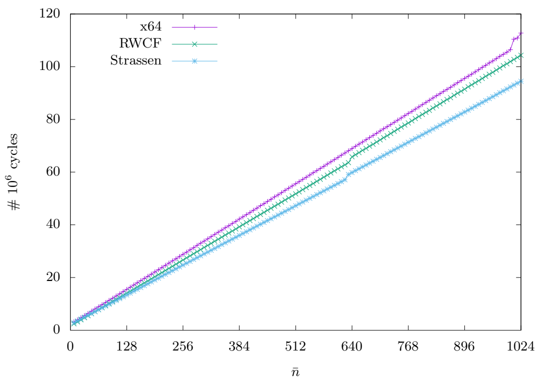
This means that for relatively small batch sizes (e.g., using a batch of only 8 computations for FrodoKEM-1344) Strassen already starts to outperform the straightforward approaches. However, the maximum observed speed-up is relatively small: a 9, 5 or 10 percent improvement for FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344. Of course, the difference between RWCF and Strassen grows with the batch size used. For even larger batch sizes it should also be checked whether applying more levels of Strassen is even faster.



(a) FrodoKEM-640-AES128 ($n = 640$)



(b) FrodoKEM-976-AES128 ($n = 976$)



(c) FrodoKEM-1344-AES128 ($n = 1344$)

Fig. 2. The performance of the row-wise cache-friendly (RWCF) and Strassen matrix multiplication (dimensions $\bar{n} \times n$ with $n \times n$) when varying $\bar{n}$.

5 Conclusions

We evaluated the performance of matrix multiplication approaches in the cryptographic setting of FrodoKEM. We consider both optimized "naïve" matrix multiplication with cubic complexity (i.e., the straightforward algorithm used in the FrodoKEM submission and the RWCF approach) as well as the Strassen multiplication algorithm (using one level).

Our results show that for the proposed parameter sets of FrodoKEM we can improve over the state-of-the-art implementation with the RWCF approach. For the matrix multiplication alone we achieve improvements up to 30 percent over the straightforward FrodoKEM approach (and are almost twice as fast when the matrix generation is pre-computed). The impact of these improvements on the full encapsulation and decapsulation operations are slightly over 20 percent. Interestingly, performing the encapsulation and decapsulation with the Strassen approach also gains improvements over the FrodoKEM approach, with an improvement of up to 12 percent for the largest parameter set. We note that the RWCF approach is to be preferred in practice.

We additionally show that for batching use-cases, where many inputs are processed at once, the Strassen approach is already to be preferred for small batches of size 8. For a practically-relevant batch size of 128 inputs the observed speed-up is in the range of 5 to 11 percent over using the efficient RWCF approach, growing with the batch size. Over the current FrodoKEM approach the improvement is even in the range of 19 to 35 percent.

This work therefore both improves on the FrodoKEM multiplication approach, and shows that the Strassen method is relevant for FrodoKEM parameter in practice.

References

1. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 5–17. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813707
2. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: 28th ACM STOC. pp. 99–108. ACM Press (May 1996). https://doi.org/10.1145/237814.237838
3. Ajtai, M., Dwork, C.: A public-key cryptosystem with worst-case/average-case equivalence. In: 29th ACM STOC. pp. 284–293. ACM Press (May 1997). https://doi.org/10.1145/258533.258604
4. Alkim, E., Bos, J.W., Ducas, L., Easterbrook, K., LaMacchia, B., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: FrodoKEM: Learning with Errors Key Encapsulations. https://github.com/microsoft/PQCrypto-LWEKE (2021)
5. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange - A new hope. In: Holz, T., Savage, S. (eds.) USENIX Security 2016. pp. 327–343. USENIX Association (Aug 2016)

6. Alman, J., Williams, V.V.: A refined laser method and faster matrix multiplication. In: Marx, D. (ed.) Symposium on Discrete Algorithms – SODA. pp. 522–539. SIAM (2021). https://doi.org/10.1137/1.9781611976465.32, https://doi.org/10.1137/1.9781611976465.32
7. Beller, M.J., Yacobi, Y.: Batch Diffie-Hellman key agreement systems and their application to portable communications. In: Rueppel, R.A. (ed.) EUROCRYPT'92. LNCS, vol. 658, pp. 208–220. Springer, Heidelberg (May 1993). https://doi.org/10.1007/3-540-47555-9_19
8. Bernstein, D.J., Brumley, B.B., Chen, M.S., Chuengsatiansup, C., Lange, T., Marotzke, A., Peng, B.Y., Tuveri, N., van Vredendaal, C., Yang, B.Y.: NTRU Prime. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
9. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU prime: Reducing attack surface at low cost. In: Adams, C., Camenisch, J. (eds.) Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10719, pp. 235–260. Springer (2017). https://doi.org/10.1007/978-3-319-72565-9_12, https://doi.org/10.1007/978-3-319-72565-9_12
10. Bos, J.W., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., Stebila, D.: Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 1006–1018. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978425
11. Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS - kyber: A cca-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy – Euro S&P. pp. 353–367. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00032
12. Bottinelli, P., Lambert, R.: Accelerating V2X cryptography through batch operations. Cryptology ePrint Archive, Report 2019/887 (2019), https://eprint.iacr.org/2019/887
13. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090262
14. Bundesamt für Sicherheit in der Informationstechnik: Cryptographic mechanisms: Recommendations and key lengths. Bsi tr-02102-1, Federal Office for Information Security (2021), https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf
15. Chen, C., Danba, O., Hoffstein, J., Hulsing, A., Rijneveld, J., Schanck, J.M., Schwabe, P., Whyte, W., Zhang, Z., Saito, T., Yamakawa, T., Xagawa, K.: NTRU. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
16. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Aho, A. (ed.) 19th ACM STOC. pp. 1–6. ACM Press (May 1987). https://doi.org/10.1145/28395.28396
17. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F.: Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 18. LNCS, vol. 10831, pp. 282–305. Springer, Heidelberg (May 2018). https://doi.org/10.1007/978-3-319-89339-6_16

18. D'Anvers, J.P., Karmakar, A., Roy, S.S., Vercauteren, F., Mera, J.M.B., Beirendonck, M.V., Basso, A.: SABER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
19. Fiat, A.: Batch RSA. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 175–185. Springer, Heidelberg (Aug 1990). https://doi.org/10.1007/0-387-34805-0_17
20. Goto, K., Geijn, R.A.v.d.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3) (2008). https://doi.org/10.1145/1356052.1356053, https://doi.org/10.1145/1356052.1356053
21. Harn, L.: Batch verifying multiple RSA digital signatures. Electronics Letters **34**, 1219–1220 (June 1998)
22. Huang, J., Smith, T.M., Henry, G.M., van de Geijn, R.A.: Strassen's algorithm reloaded. In: West, J., Pancake, C.M. (eds.) International Conference for High Performance Computing, Networking, Storage and Analysis – SC. pp. 690–701. IEEE Computer Society (2016). https://doi.org/10.1109/SC.2016.58, https://doi.org/10.1109/SC.2016.58
23. Intel: Advanced vector extensions programming reference. https://software.intel.com/content/dam/develop/external/us/en/documents/36945 (2011)
24. Karati, S., Das, A.: Faster batch verification of standard ECDSA signatures using summation polynomials. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 14. LNCS, vol. 8479, pp. 438–456. Springer, Heidelberg (Jun 2014). https://doi.org/10.1007/978-3-319-07536-5_26
25. Karati, S., Das, A., Chowdhury, D.R., Bellur, B., Bhattacharya, D., Iyer, A.: Batch verification of ECDSA signatures. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 12. LNCS, vol. 7374, pp. 1–18. Springer, Heidelberg (Jul 2012)
26. Karati, S., Das, A., Chowdhury, D.R., Bellur, B., Bhattacharya, D., Iyer, A.: New algorithms for batch verification of standard ECDSA signatures. Journal of Cryptographic Engineering **4**(4), 237–258 (Nov 2014). https://doi.org/10.1007/s13389-014-0082-x
27. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**, 203–209 (1987)
28. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (2015)
29. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (May / Jun 2010). https://doi.org/10.1007/978-3-642-13190-5_1
30. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO'85. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (Aug 1986). https://doi.org/10.1007/3-540-39799-X_31
31. Naehrig, M., Alkim, E., Bos, J., Ducas, L., Easterbrook, K., LaMacchia, B., Longa, P., Mironov, I., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D.: FrodoKEM. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
32. National Institute of Standards and Technology: Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization
33. Peikert, C.: A decade of lattice cryptography. Foundations and Trends in Theoretical Computer Science **10**(4), 283–424 (2016). https://doi.org/10.1561/0400000074, https://doi.org/10.1561/0400000074

34. Peikert, C., Regev, O., Stephens-Davidowitz, N.: Pseudorandomness of ring-LWE for any ring and modulus. In: Hatami, H., McKenzie, P., King, V. (eds.) 49th ACM STOC. pp. 461–473. ACM Press (Jun 2017). https://doi.org/10.1145/3055399.3055489
35. Proos, J., Zalka, C.: Shor's discrete logarithm quantum algorithm for elliptic curves. Quantum Inf. Comput. **3**, 317—344 (2003), https://cds.cern.ch/record/602816
36. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th ACM STOC. pp. 84–93. ACM Press (May 2005). https://doi.org/10.1145/1060590.1060603
37. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the Association for Computing Machinery **21**(2), 120–126 (1978)
38. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
39. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: 35th FOCS. pp. 124–134. IEEE Computer Society Press (Nov 1994). https://doi.org/10.1109/SFCS.1994.365700
40. Smith, T.M., van de Geijn, R.A., Smelyanskiy, M., Hammond, J.R., Zee, F.G.V.: Anatomy of high-performance many-threaded matrix multiplication. In: IEEE International Parallel and Distributed Processing Symposium. pp. 1049–1059. IEEE Computer Society (2014). https://doi.org/10.1109/IPDPS.2014.110, https://doi.org/10.1109/IPDPS.2014.110
41. Strassen, V.: Gaussian elimination is not optimal. Numerische mathematik **13**(4), 354–356 (1969)
42. Zee, F.G.V., van de Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Trans. Math. Softw. **41**(3), 14:1–14:33 (2015). https://doi.org/10.1145/2764454, https://doi.org/10.1145/2764454