Optimised Lattice-Based Key Encapsulation in Hardware

James Howe[1], Marco Martinoli[2] Elisabeth Oswald[2], and Francesco Regazzoni[3]

[1] PQShield Ltd., Oxford, United Kingdom
james.howe@pqshield.com,
[2] Department of Computer Science, University of Bristol, United Kingdom
{marco.martinoli,elisabeth.oswald}@bristol.ac.uk
[3] Advanced Learning and Research Institute, Università della Svizzera Italiana, Switzerland
regazzoni@alari.ch

Abstract. Recently, a standard lattice-based key encapsulation mechanism has been shown to have promising performance on FPGA. The cryptographic scheme, named FrodoKEM, is one of many candidates for NIST's call for post-quantum standardisation. This research proposes optimised designs for FrodoKEM, concentrating on high throughput by paralleling the matrix multiplication operations within the cryptographic scheme. This process is eased by the use of a much smaller and faster PRNG. The parallelisations proposed also complement the addition of first-order masking to the decapsulation module. Overall, we significantly increase the throughput of FrodoKEM, achieving up to 840 key generations per second, 825 encapsulations per second, and 763 decapsulations per second, with almost no impact on the FPGA area consumption compared to the previous state-of-the-art.

1 Introduction

The future development of a scalable quantum computer will allow us to solve in, polynomial time, several problems which are considered intractable for classical computers. Certain fields, such as biology and physics, would certainly benefit from this "quantum speed up", however this could be disastrous for security. The security of our current public-key infrastructure is based on the computational hardness of the integer factorization problem (RSA) and the discrete logarithm problem (ECC). These problems, however, will be solved in polynomial time by a machine capable of executing the Shor's algorithm [9].

To promptly react to the threat, the scientific community started to study, propose, and implement public-key algorithms, to be deployed on classical computers, but based on problems computational difficult to solve also using a quantum computer or a classical computer. This effort is supported by governmental and standardisation agencies, which are pushing for new and quantum resistant algorithms. The most notable example of these activities is the open contest that NIST [6] is running for the selection of the next public-key standardised

algorithms. The contest started at the end of 2017 and is expected to run for 5 to 7 years.

Approximately seventy algorithms were submitted to the standardisation process, with the large majority of them being based on the hardness of lattice problems. Lattice-based cryptographic algorithms are a class of algorithms which base their security on the hardness of problems such has finding the shortest non-zero vector in a lattice. The reason for such a large number of candidates is because lattice-based algorithms are extremely promising: they can be implemented efficiently and they are extremely versatile, allowing to efficiently implement cryptographic primitives such as, digital signatures, key encapsulation, and identity-based encryption.

As in the past case for standardising AES and SHA-3, the parameters which will be used for selection include the security of the algorithm and its efficiency when implemented in hardware and software. NIST have also stated that algorithms which can be made robust against physical attacks in an effective and efficient way will be preferred [7]. Thus, it is important, during the scrutiny of the candidates, to explore the potential of implementing these algorithms on a variety of platforms, and to assess the overhead of adding countermeasures against physical attacks.

To this end, this paper concentrates on FrodoKEM, a key encapsulation algorithm submitted to NIST as a potential post-quantum standard. FrodoKEM is a conservative candidates due to its hardness being based on standard lattices, as opposed to Ring-LWE or Module-LWE, as such its has had limited practical evaluations. Thus, we explore the possibility to efficiently implementing it in hardware and we estimate the overhead of protecting against power analysis attacks using first-order masking. To maximise the throughput, while maintaining the area occupation minimal, we rely on a parallelised implementation of the matrix multiplication. To be parallelised, however, the matrix multiplication requires the use of a smaller and more performant random number generator. We propose to achieve the performance required for the PRNG by using Trivium, which we used instead of AES or (c)SHAKE.

The rest of the paper is organised as follows. Section 2 discusses the background and the related works. Section 3 introduces the proposed hardware architectures and the main design decisions. Section 4 reports the results obtained while synthesising our design on reconfigurable hardware and compares our performance against the state-of-the-art. We conclude the paper in Section 5.

2 Background and Related Work

In this section we summarise Frodo and its efficient implementations and we recall the principles of masking.

2.1 Implementations of Frodo

FrodoKEM [5] is a key encapsulation mechanism (KEM) based on the original standard lattice problem Learning With Errors (LWE) [8]. FrodoKEM is a fam-

ily of IND-CCA secure KEMs, the structure of which is based on a key exchange variant FrodoCCS [1]. FrodoKEM comes with two parameter sets FrodoKEM-640 and FrodoKEM-976, a summary of which is shown in Table 1. FrodoKEM key generation is shown in Algorithm 1, encapsulation is shown in Algorithm 2, and decapsulation is shown in Algorithm 3. The most computationally heavy operations in FrodoKEM are in Line 7 of Algorithm 1, Line 7 of Algorithm 2, and Line 11 of Algorithm 3, that is the matrix multiplication of two matrices , sampled from the error sampler and PRNG, respectively. The LWE instance is then completed by adding an additional error from the error sampler. This is followed by a much smaller LWE operation, in which a random key is encoded. Finally, these ciphertexts are used to calculate a shared secret (**ss**) via (c)SHAKE. The matrices generated heavily utilise PRNGs, suggested by the authors via AES or (c)SHAKE. The output of these algorithms have nice statistical properties, but the overhead required to achieve this is high.

Table 1: Implemented FrodoKEM parameter sets.

	FrodoKEM-640	FrodoKEM-976
Matrix Dimensions	$n = 640, \bar{n} = \bar{m} = 8$	$n = 976, \bar{n} = \bar{m} = 8$
Modulus (q)	$2^{15} = 32768$	$2^{16} = 65536$
Distribution (χ)	$\sigma = 2.8$	$\sigma = 2.3$
Security	128 bits	192 bits

Algorithm 1 FrodoKEM key pair generation

1: **procedure** KeyGen(1^ℓ)
2: Generate random seeds $\mathbf{s}||\text{seed}_\mathbf{E}||\mathbf{z} \leftarrow_\$ U(\{0,1\}^{128})$
3: Generate pseudo-random seed $\text{seed}_\mathbf{A} \leftarrow H(\mathbf{z})$
4: Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow \text{Frodo.Gen}(\text{seed}_\mathbf{A})$
5: $\mathbf{S} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_\mathbf{E}, n, \bar{n}, T_\chi, 1)$
6: $\mathbf{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_\mathbf{E}, n, \bar{n}, T_\chi, 2)$
7: Compute $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
8: **return** public key $pk \leftarrow \text{seed}_\mathbf{A}||\mathbf{B}$ and secret key $sk' \leftarrow (\mathbf{s}||\text{seed}_\mathbf{A}||\mathbf{B}, \mathbf{S})$
9: **end procedure**

Naehrig et al. [5] report the results of the implementation on a 64-bit ARM Cortex-A72 (with the best performance achieved by using OpenSSL AES implementation, that benefits from the NEON engine) and an Intel Core i7-6700 (x64 implementation using AVX2 and AES-NI instructions). Employing modular arithmetic ($q \leq 2^{16}$) results in using efficient and easy to implement single-precision arithmetic. The sampling of the error term (16 bits per sample) is done

Algorithm 2 FrodoKEM encapsulation

procedure ENCAPS($pk = \text{seed}_{\mathbf{A}}||\mathbf{b}$)
  Choose a uniformly random key $\mu \leftarrow U(\{0,1\}^{\text{len}_\mu})$
  Generate pseudo-random values $\text{seed}_{\mathbf{E}}||\mathbf{k}||\mathbf{d} \leftarrow G(pk||\mu)$
  Sample error $\mathbf{S}' \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_{\mathbf{E}}, \bar{m}, n, T_\chi, 4)$
  Sample error $\mathbf{E}' \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_{\mathbf{E}}, \bar{m}, n, T_\chi, 5)$
  Generate $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow \text{Frodo.Gen}(\text{seed}_{\mathbf{A}})$
  Compute $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
  Compute $\mathbf{c}_1 \leftarrow \text{Frodo.Pack}(\mathbf{B}')$
  Sample error $\mathbf{E}'' \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_{\mathbf{E}}, \bar{m}, \bar{n}, T_\chi, 6)$
  Compute $\mathbf{B} \leftarrow \text{Frodo.Unpack}(\mathbf{b}, n, \bar{n})$
  Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
  Compute $\mathbf{C} \leftarrow \mathbf{V} + \text{Frodo.Encode}(\mu)$
  Compute $\mathbf{c}_2 \leftarrow \text{Frodo.Pack}(\mathbf{C})$
  Compute $\mathbf{ss} \leftarrow F(\mathbf{c}_1||\mathbf{c}_2||\mathbf{k}||\mathbf{d})$
  return ciphertext $\mathbf{c}_1||\mathbf{c}_2||\mathbf{d}$ and shared secret $\mathbf{ss}$
end procedure

by inversion sampling using a small LUT corresponds to the discrete cumulative density functions (CDT sampling).

There has been a number of software and hardware optimisations of FrodoKEM. Howe et al. [4] report both software and hardware designs for microcontroller and FPGA. The hardware design focuses on a plain implementation by using only one multiplier in order to fairly compare with previous work and the proposed software implementation. Due to their use of cSHAKE for PRNG, they have to pre-store a lot of the randomness into BRAM and then constantly update these values. Due to this, the implementations do not have the ability to increase the number of multipliers and uses large amounts of memory. So far there has been no investigation of side-channel analysis for FrodoKEM other than ensuring the implementations run in constant-time [4].

2.2 Side-Channel Analysis

In their call for proposals, NIST specified that algorithms which can be protected against side-channel attacks in an effective and efficient way are to be preferred [7]. To provide a whole picture about the performance of a candidate, it is thus important to evaluate also the cost of implementing "standard" countermeasures against these attacks.

In FrodoKEM specifications, cache and timing attacks can be mitigated using well known guidelines for implementing the algorithm. For timing attacks, these include to avoiding use of data derived from the secret to access the addresses and in conditional branches. To counteract cache attacks it is necessary to ensure that all the operations depending on secrets are executed in constant-time.

Power analysis attacks can be addressed using masking. Masking is one of the most widespread and better understood techniques to protect against passive side-channel attacks. In its most basic form, a mask is drawn uniformly from

Algorithm 3 The FrodoKEM decapsulation

procedure DECAPS($sk = (\mathbf{s}||\text{seed}_\mathbf{A}||\mathbf{b}, \mathbf{S}), \mathbf{c}_1||\mathbf{c}_2||\mathbf{d}$)
  Compute $\mathbf{B}' \leftarrow$ Frodo.Unpack($\mathbf{c}_1$)
  Compute $\mathbf{C} \leftarrow$ Frodo.Unpack($\mathbf{c}_2$)
  Compute $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
  Compute $\mu' \leftarrow$ Frodo.Decode($\mathbf{M}$)
  Parse $pk \leftarrow \text{seed}_\mathbf{A}||\mathbf{b}$
  Generate pseudo-random values $\text{seed}'_\mathbf{E}||\mathbf{k}'||\mathbf{d}' \leftarrow G(pk||\mu')$
  Sample error $\mathbf{S}' \leftarrow$ Frodo.SampleMatrix($\text{seed}'_\mathbf{E}, \bar{m}, n, T_\chi, 4$)
  Sample error $\mathbf{E}' \leftarrow$ Frodo.SampleMatrix($\text{seed}'_\mathbf{E}, \bar{m}, n, T_\chi, 5$)
  Generate $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow$ Frodo.Gen($\text{seed}_\mathbf{A}$)
  Compute $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
  Sample error $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix($\text{seed}'_\mathbf{E}, \bar{m}, n, T_\chi, 6$)
  Compute $\mathbf{B} \leftarrow$ Frodo.Unpack($\mathbf{b}, n, \bar{n}$)
  Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
  Compute $\mathbf{C}' \leftarrow \mathbf{V} +$ Frodo.Encode(μ')
  if $\mathbf{B}'||\mathbf{C} = \mathbf{B}''||\mathbf{C}'$ and $\mathbf{d} = \mathbf{d}'$ **return ss** $\leftarrow F(\mathbf{c}_1||\mathbf{c}_2||\mathbf{k}'||\mathbf{d})$
  else return ss $\leftarrow F(\mathbf{c}_1||\mathbf{c}_2||\mathbf{s}||\mathbf{d})$
end procedure

random and added to the secret. The resulting masked value, which is effectively a one-time-pad, and the mask are jointly called *shares*: if taken singularly they are statistically independent from the secret, and they must be combined to obtain the secret back. Any operation that previously involved the secret has to be turned into an operation over its shares. As long as they are not combined, any leakage from them will be statistically independent of the secret too. In our context, we show how masking can easily applied to FrodoKEM at a very low cost. We therefore argue the overhead that a protected implementation of Frodo in hardware incurs is minimal, hence making it a strong candidate when side-channel analysis are a concern. The reason behind this is that the only operation using the secret matrix $\mathbf{S}$ is the computation of the matrix $\mathbf{M}$ as $\mathbf{C} - \mathbf{B}'\mathbf{S}$ during decapsulation. When $\mathbf{S}$ is split in two (or more) shares using addition modulo q, the above multiplication by $\mathbf{B}'$ can be simply applied to both shares independently. Results are then subtracted by $\mathbf{C}$ one-by-one, so that computations never depend on both shares simultaneously.

3 Hardware Design

Our main design goal is to improve the throughput of the lattice-based key encapsulation scheme FrodoKEM [5] when implemented in hardware. As described in Section 2, FrodoKEM is one of the leading conservative candidates submitted to the NIST post-quantum standardisation effort [6]. Moreover, it has been shown to have appealing qualities which make it an ideal candidate for hardware implementations, such as having a power-of-two modulus and significantly easier parameter selection. However a complete exploration of the possible hardware

optimisations applicable to FrodoKEM is yet to come. For instance, previous implementations do not consider parallelisation or other design alternatives capable of significantly improve the throughput.

As described in Section 2, FrodoKEM requires heavy use of PRNGs. In the algorithm specifications it is suggest to either use (c)SHAKE or AES. In particular, the most computationally intensive operations, Line 7 of Algorithm 2, requires $n \times n$ (for $n = 640$ or 976) 16-bit pseudo-random values. To not be the bottle-neck, PRNG needs to achieve a high throughput, typically in the range of 16 bits per clock cycle. In a previous hardware design, proposed by Howe et al. [4], high throughput for the PRNG was achieved by pre-calculating randomness and storing it in BRAM. Random data newly calculated was then written into the memory, overwriting the random data previously stored. This is an efficient approach, however a more efficient PRNG that would not require BRAM usage, would have the potential to increase the operating frequency of the design and thus improve its throughput.

Another issue with the use of AES or (c)SHAKE is the relatively large area overhead. For example, cSHAKE used within FrodoKEM-640 Encaps occupies 42% of the overall hardware resources [4]. Bos et al. [2] recently improved the throughput of software implementations of FrodoKEM by leveraging a different PRNG; xoshiro128**. To improve the parallelism of our implementation, we put further this idea to hardware and replace the suggested PRNG. We explored several options and we decided to integrate into our design an unrolled x32 Trivium [3] module. This is compatible with the security requirements of the FrodoKEM submission. In fact, the authors of the algorithm suggests that replacing the PRNG with another, that still has good statistical pseudo-random properties, still guarantees the security claims of FrodoKEM. The Trivium architecture we integrate has high throughput and maintains the cryptographic security required in the FrodoKEM specifications, thus perfectly fits our needs.

3.1 Hardware Optimisations

In order to fully explore the potential of FrodoKEM in hardware, we propose several architectures characterised by different design goals (in terms of throughput). We use the proposed architecture to implement key generation, encapsulation, and decapsulation, on both sets of parameters proposed in the specifications: FrodoKEM-640 and FrodoKEM-976. Our designs uses 1x, 4x, 8x, and 16x parallel multiplications during the most computationally intensive parts in FrodoKEM. The following is the LWE calculation of the type:

$$\mathbf{B} = \mathbf{SA} + \mathbf{E}, \tag{1}$$

required in key generation, encapsulation, and decapsulation. It takes approximately 97.5% of the overall computations [4]. As in literature, we exploit DSP slices on the FPGA for the multi-and-accumulate (MAC) operations required for matrix multiplication. Hence, each parallel multiplication of the proposed designs uses its own DSP slice.

The LWE matrix multiplication component incurs in a large computational overhead. Because of this, it is a nice target for optimisations. Our optimisations heavily rely on parallelisation. Firstly we describe the basic LWE multiplier, that includes just one multiplication component. Then we describe how this core is parallelised, allowing us to significantly improve the throughput.

The LWE core is essentially made by vector-matrix multiplication (that is, $\mathbf{S}[\text{row}] \times \mathbf{A}$), addition of an error (that is, $\mathbf{E}$), and, when needed, an addition of the encoding of message data. Since the matrix $\mathbf{S}$ consists of a large number of column entries (either 640 or 976) but only 8 row entries (for both parameter sets), we decided to implement a vector-matrix multiplier, instead of matrix-matrix one. By doing this, we can reuse the same hardware architecture for each row of $\mathbf{S}$, saving significant hardware resources. Each run of the row-column MAC operation exploits a DSP slice on the FPGA, which fits within the 48-bit MAC size of the FPGA. The DSP slice is ideal for these operations, but it also ensures constant computational time, since each multiplication requires one clock cycle. Once each row-column MAC operation is completed, an error value is added from the CDT sampler. These values are consistently added into an instantiation of SHAKE, which is required to calculate the shared secret, as well as being output as the ciphertext. This process is pipelined to ensure high throughput and constant runtime. A high-level overview of the whole architecture is shown in Figure 1.

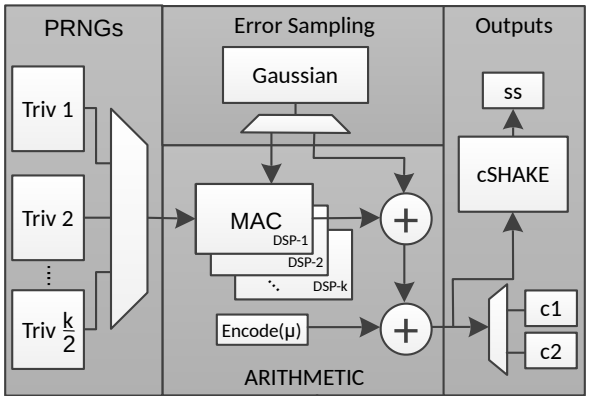

Fig. 1: A high-level overview of the proposed hardware designs for FrodoKEM for k parallel multipliers.

To avoid to use BRAM and while keeping the throughput needed by the MAC operations of the matrix multiplications, the designs require 16 bits of pseduo-randomness per multiplication per clock cycle. Thus, for every two parallel multiplications we require one Trivium instantiation, whose 32-bit output per clock cycle is split up to form two 16-bit pseudo-random integers. This pseudo-randomness forms the matrix $\mathbf{A}$ in Equation 1, whereas the matrix $\mathbf{S}$

and **E** require randomness taken from a Gaussian-like distribution. The cumulative distribution table (CDT) sampler technique has been shown to be the most suitable one for hardware. However compared with previous works, we replace the use of AES as a psuedo-random input with Trvium. This ensures the same high throughput, but requires significantly less area on the FPGA.

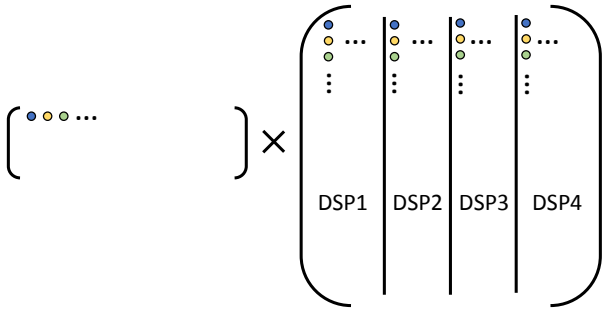

Fig. 2: Parallelising matrix multiplication, for $\mathbf{S} \times \mathbf{A}$, used within LWE computations for an example of $k = 4$ parallel multiplications.

Overall, the technique we use to parallelise Equation 1 is to vertically partition the matrix **A** into k equal sections, where k is the number of parallel multiplications used. This is shown in Figure 2 for $k = 4$ parallel multiplications, utilising 4 DSP slices for MAC. Each vector on the LHS of Figure 2 remains the same for each of the k operations. We repeat this vector-matrix operation for the $\bar{n} = 8$ rows of the matrix **S**. This technique is used across all designs for the three cryptographic modules to ensure consistency.

In order to produce enough randomness for these multiplications to have no delays, we need one instance of our PRNG, Trivium, for every two parallel multiplications. This because each element of the matrix **A** is set to be a 16-bit integer and each output from Trivium is 32 bits, that is, two 16-bit integers.

3.2 Efficient First-Order Masking

We implement first-order masking to the decapsulation operation $\mathbf{M} = \mathbf{C} - \mathbf{B}'\mathbf{S}$, as this is the only instance where secret-key information is used. Our design allows to implement this masking schema without affecting the area consumption or throughput. This is achieved by re-using the optimisations previously discussed. The matrix **S** is split using the same technique from Figure 2 and our secret shares are generated by re-using the Trivium instances. By computing these calculations in parallel, the masked calculation of **M** has the same runtime as the one needed to complete the calculation when masking is not used.

4 Results

In this section we presents the results obtained when implementing our FrodoKEM architecture. The first analysis is directed towards the performance of the PRNG. When compared to cSHAKE, the PRNG previously used in literature, Trivium (the PRNG we propose to use), occupies 4.5x less FPGA slices. This means that when we instantiate a higher number of parallel multipliers, we consume far less FPGA area than what would be needed when using cSHAKE as discussed in the algorithm proposal. The increase in area occupation due to parallel implementation is essentially the only reason for area increase when we move from a base designs to a design of the same module with a higher number of parallel multipliers. This is because the vector being multiplied remains constant, we just require some additional registers to store these extra random elements. Additionally, we are able to use a much smaller version of SHA-3 for generating the random seeds (< 400 FPGA slices) and shared secrets as the computational requirements for it have significantly decreased.

There is a significant increase in area consumption of all the decapsulation results which do not utilise BRAM. This is mainly due to the need of storing public-key and secret-key matrices. We provide results for both architectures with and without BRAM. The design without BRAM has a significantly higher throughput, due to the much higher frequency. These results are reported in Figure 4, which shows the efficiency of each design (namely their throughput) per FPGA slice utilised. Figure 3 shows a slice count summary of all the proposed designs, showing a consistent and fairly linear increase in slice utilisation as the number of parallel multipliers increases. We note on decapsulation results in Figure 3 where the results would lie if BRAM is used, hence the total results for without BRAM include both red areas. In most cases slice counts at least double for decapsulation when BRAM is removed, with only slight increases in throughout, hence it might be prudent in some use cases to keep BRAM usage.

Compared to the previous works, we show significant savings in FPGA area resource consumption. For instance, comparing to FrodoKEM module [4] (that is, using one multiplier) we reduce slice consumption by 3.6x and 5.4x for key generation and 1.6x for encapsulation, all whilst not requiring any BRAM, whereas previous results utilise BRAM. For decapsulation, we save between 1.6x and 2.6x slices when BRAM is used and gain in slice counts by 1.5x and 1.1x if BRAM is not used. This increase is expected since more than half of this is due to storage otherwise used in BRAM.

Since the majority of the proposed designs operate without BRAM, we were able to attain a much higher frequency than previous works. Overall our throughput outperforms previous comparable results, by factors between 1.13x and 1.19x [4]. Moreover, whilst maintaining less area consumption than previous research we were able to increase the amount of parallel multipliers used. As a result, we can achieve up to 840 key generations per second (a 16.5x increase), 825 encapsulations per second (a 16.2x increase), and 710 operations per second (a 15.6x increase).

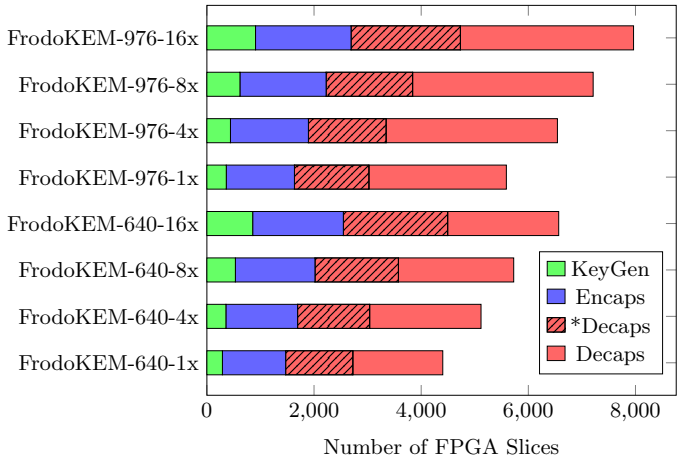

Fig. 3: Visualisation of FPGA slice consumption of FrodoKEM's key generation, encapsulation, & decapsulation on a Xilinx Artix-7. Decaps values overlap to show results with (*) and without BRAM.

We also maintain the constant runtime which the previous implementation attains, as well as implementing first-order masking during decapsulation. The clock cycle counts for each module are easy to calculate; key generation requires $(n^2\bar{n})/k$ clocks, encapsulation requires $(n^2\bar{n} + \bar{n}^2 n)/k$ clocks, and decapsulation requires $(n^2\bar{n} + 2\bar{n}^2 n)/k$ clocks, for dimensions $n = 640$ or 976, $\bar{n} = 8$, and k referring to the number of parallel multipliers used.

5 Conclusions

The main contributions of this research is to evaluate the lattice-based key encapsulation mechanism and potential NIST post-quantum standard, FrodoKEM [5], in hardware. We develop designs which can reach up to 825 operations per second, where most of the designs fit in under 1500 slices. We significantly improve the state of the art by increasing the number of parallel multipliers we use during matrix multiplication. In order to do this efficiently, we replace the inefficient PRNG previously used, cSHAKE, with a much faster and smaller PRNG, Trivium. As a result, we are able to attain significantly higher throughput efficiency compared to previous research. Our implementations also run in constant computational time and the designs comply with the Round 2 version of FrodoKEM in all aspects except for this PRNG choice. To further evaluate the performance of FrodoKEM, we implemented first-order masking for decapsulation, and we showed that it can be achieved with almost no effect on performance.

The results show that FrodoKEM is an ideal candidate for hardware designs, showing potential for high-throughput performances whilst still maintaining relatively small FPGA area consumption. Moreover, compared to other NIST

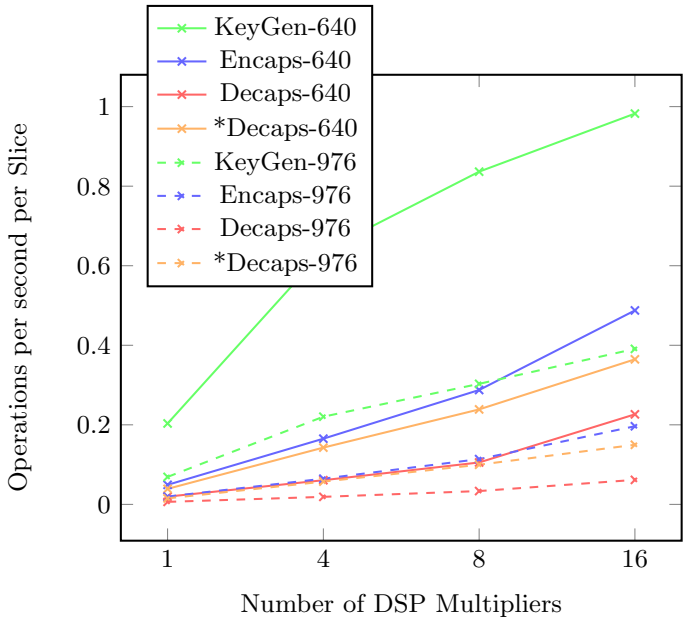

Fig. 4: Comparison of the throughput performance per FPGA slice on a Xilinx Artix-7.

lattice-based candidates, it has a lot more flexibility, such as increasing throughput without completely re-designing the multiplication component, compared to, for example, a NTT multiplier.

References

1. 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: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 1006–1018 (2016)
2. Bos, J.W., Friedberger, S., Martinoli, M., Oswald, E., Stam, M.: Fly, you fool! faster frodo for the arm cortex-m4. Cryptology ePrint Archive, Report 2018/1116 (2018), https://eprint.iacr.org/2018/1116
3. De Canniere, C., Preneel, B.: Trivium. In: New Stream Cipher Designs, pp. 244–266. Springer (2008)
4. Howe, J., Oder, T., Krausz, M., Güneysu, T.: Standard lattice-based key encapsulation on embedded devices. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 372–393 (2018)
5. 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 (2017), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions

6. NIST: Post-quantum crypto project. http://csrc.nist.gov/groups/ST/post-quantum-crypto/ (2016)
7. NIST: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf (2016)
8. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005. pp. 84–93 (2005). https://doi.org/10.1145/1060590.1060603, http://doi.acm.org/10.1145/1060590.1060603
9. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (Oct 1997)

Table 2: FPGA resource consumption of the proposed FrodoKEM hardware designs, using Trivium as a PRNG, with 1, 4, 8, or 16 parallel multipliers and also using both parameter sets FrodoKEM-640 and FrodoKEM-976. Results with BRAM usage have an asterisk (*). Also shown are the hardware results of Trivium and the error sampler. All results utilise a Xilinx Artix-7 FPGA

FrodoKEM Protocol	**LUT/FF**	**Slices**	**DSP/ BRAM**	**MHz**	**Ops/ Sec**
KeyGen-640 1x	971/433	290	1/0	191	59
KeyGen-640 4x	1174/781	355	4/0	185	226
KeyGen-640 8x	1679/1570	532	8/0	182	445
KeyGen-640 16x	2587/2994	855	16/0	172	840
Encaps-640 1x	4246/2131	1180	1/0	190	58
Encaps-640 4x	4620/2552	1338	4/0	183	221
Encaps-640 8x	5155/3356	1485	8/0	177	427
Encaps-640 16x	5796/4694	1692	16/0	171	825
Decaps-640 1x	10518/2299	2933	1/0	190	57
Decaps-640 4x	11581/2818	3424	4/0	174	208
Decaps-640 8x	13128/3737	3710	8/0	164	391
Decaps-640 16x	14528/5335	4020	16/0	160	763
*Decaps-640 1x	4466/2152	1254	1/12.5	162	49
*Decaps-640 4x	4841/2661	1345	4/12.5	161	192
*Decaps-640 8x	5476/3479	1558	8/12.5	156	372
*Decaps-640 16x	6881/5081	1947	16/12.5	149	710
KeyGen-976 1x	1243/441	362	1/0	189	25
KeyGen-976 4x	1458/792	440	4/0	184	97
KeyGen-976 8x	1967/1576	617	8/0	178	187
KeyGen-976 16x	2869/3000	908	16/0	169	355
Encaps-976 1x	4650/2118	1272	1/0	187	25
Encaps-976 4x	4996/2611	1455	4/0	180	94
Encaps-976 8x	5562/3349	1608	8/0	175	183
Encaps-976 16x	6188/4678	1782	16/0	168	350
Decaps-976 1x	14217/2295	3956	1/0	188	25
Decaps-976 4x	16234/2853	4648	4/0	170	88
Decaps-976 8x	17451/3687	4985	8/0	161	167
Decaps-976 16x	18960/5285	5274	16/0	157	325
*Decaps-640 1x	4888/2153	1390	1/19	162	21
*Decaps-640 4x	5259/2662	1450	4/19	160	83
*Decaps-640 8x	5888/3490	1615	8/19	155	161
*Decaps-640 16x	7213/5087	2042	16/19	148	306
Error+Trivium	401/311	179	0/0	211	211m
Trivium	296/299	169	0/0	220	220m
KeyGen-640 [4]	3771/1800	1035	1/6	167	51
Encaps-640 [4]	6745/3528	1855	1/11	167	51
Decaps-640 [4]	7220/3549	1992	1/16	162	49
KeyGen-976 [4]	7139/1800	1939	1/8	167	22
Encaps-976 [4]	7209/3537	1985	1/16	167	22
Decaps-976 [4]	7773/3559	2158	1/24	162	21