Noname manuscript No.
(will be inserted by the editor)

Exploring Parallelism to Improve the Performance of FrodoKEM in Hardware

James Howe · Marco Martinoli · Elisabeth Oswald · Francesco Regazzoni

Received: 02 April 2020 / Accepted: 12 February 2021

Abstract FrodoKEM is a lattice-based key encapsulation mechanism, currently a semi-finalist in NIST's post-quantum standardization effort. A condition for these candidates is to use NIST standards for sources of randomness (i.e., seed-expanding), and as such most candidates utilize SHAKE, an XOF defined in the SHA-3 standard. However, for many of the candidates, this module is a significant implementation bottleneck. Trivium is a lightweight, ISO standard stream cipher which performs well in hardware and has been used in previous hardware designs for lattice-based cryptography. This research proposes optimized designs for FrodoKEM, concentrating on high throughput by parallelising the matrix multiplication operations within the cryptographic scheme. This process is eased by the use of Trivium due to its higher throughput and lower area consumption. The parallelisations proposed also complement the addition of first-order masking to the decapsulation module. Overall, we significantly increase the throughput of FrodoKEM; for encapsulation we see a 16x speed-up, achieving 825 operations per second, and for decapsulation we see a 14x speed-up, achieving 763 operations per second, compared to the previous state-of-the-art, whilst also maintaining a similar FPGA area footprint of less than 2000 slices.

J. Howe
PQShield Ltd., Oxford, United Kingdom.
E-mail: james.howe@pqshield.com.

M. Martinoli
Proton Technologies AG, Geneva, Switzerland.
E-mail: marco.martinoli@protonmail.com.

E. Oswald
Department of Computer Science, University of Bristol, United Kingdom and University of Klagenfurt, Austria.
E-mail: elisabeth.oswald@bristol.ac.uk and E-mail: elisabeth.oswald@aau.at.

F. Regazzoni
University of Amsterdam, The Netherlands and Advanced Learning and Research Institute, Università della Svizzera Italiana, Switzerland.
E-mail: f.regazzoni@uva.nl and E-mail: regazzoni@alari.ch.

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 Shor's algorithm [29].

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 computationally difficult to solve also using a quantum or classical computer. This effort is supported by governmental and standardization agencies, which are pushing for new and quantum resistant algorithms. The most notable example of these activities is the open contest that NIST [20] is running for the selection of the next public-key standardized 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 standardization 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 as 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 standardizing 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 [21]. 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.

To this end, this paper concentrates on FrodoKEM, a key encapsulation mechanism submitted to NIST as a potential post-quantum standard. FrodoKEM is a conservative candidate due to its hardness being based on standard lattices, as opposed to Ring/Module-LWE, thus having limited practical evaluations. Thus, we explore the possibility to efficiently implementing it in hardware and estimate the overhead of protection against power analysis attacks using first-order masking. To maximize the throughput, we rely on a parallelised implementations of the matrix multiplication. Although we do not utilise specialised techniques for parallelising the matrix multiplication, there exists a lot of prior art in this area of research [14,24]. We also aim to have a relatively low FPGA area consumption. To be parallelised, however, the matrix multiplication requires the use of a smaller and more performant pseudo-random number generator. We propose to achieve the performance required for the randomness generation by using Trivium, an international standard under ISO/IEC 29192-3 [13] and selected as part of the eSTREAM project, specifically selected for its hardware performance[1]. We utilize this instead of AES or SHAKE, as per the FrodoKEM specifications. We do this as a design exploration study and not (per se) as a recommendation; other alternative ciphers or hash functions with similar security arguments and performance profiles in hardware could equally be applied.

The rest of the paper is organized 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 synthesizing our design on reconfigurable hardware and compares our performance against the state-of-the-art. We conclude the paper in Section 5.

[1] `https://www.ecrypt.eu.org/stream/e2-trivium.html`

2 Background and Related Work

In this section we provide some background on previous hardware implementations post-quantum cryptographic schemes, focusing on those which are candidates of NIST's standardization effort. We will also elaborate more on FrodoKEM and its implementations as well as recalling the principles of masking.

2.1 Previous post-quantum hardware implementations

In order to provide a reference point on the state-of-the-art in hardware designs of post-quantum candidates, we provide a brief summary here. Table 1 shows the area and throughput performances of candidates, separated by their post-quantum hardness type. Firstly, it is quite clear that SIKE is the largest and slowest of the schemes, consuming quite a large portion of the (expensive) FPGA they benchmark on. Hash-based and code-based schemes on the other hand, whilst requiring similarly large FPGA resources, makes up for this and provides a high throughput. Lattice-based schemes generally enjoy the best-of-both-worlds in terms of area consumption and performance, having a relatively small FPGA area consumption and a relatively high throughput. Not only is this seen in Table 1, but this is also true for other lattice-based schemes, pre NIST's post-quantum competition. Within the lattice-based candidates, the ideal lattice schemes are, as expected, much more efficient in terms area throughput performance compared to standard lattices. This is essentially because of the complexity of their respective multiplications; in standard lattice schemes the matrix multiplications have $\mathcal{O}(n^2)$ complexity, whereas ideal and module schemes are able to use a NTT polynomial multiplier, reducing the complexity to $\mathcal{O}(n \log n)$.

2.2 Implementations of FrodoKEM

FrodoKEM [19] is a key encapsulation mechanism (KEM) based on the original standard lattice problem learning with errors (LWE) [25]. FrodoKEM is a family of IND-CCA secure KEMs, the structure of which is based on a key exchange variant FrodoCCS [7]. FrodoKEM comes with two parameter sets FrodoKEM-640 and FrodoKEM-976, a summary of which is shown in Table 2. FrodoKEM

Table 1: A summary of the current state-of-the-art of hardware designs of NIST post-quantum candidates, implemented on FPGA.

	Crypto. Implementation	Device	LUT	FF	Slice	DSP	BRAM	MHz	Ops/Sec
Code	Niederreiter KeyGen [30]	Stratix-V	–	–	39,122	–	827	230	75
	Niederreiter Encrypt [30]	Stratix-V	–	6,977	4,276	–	0	448	50,000
	Niederreiter Decrypt [30]	Stratix-V	–	48,050	20,815	–	88	290	12,500
Isogeny	SIKE 3-cores (Total) [17]	Virtex-7	27,713	38,489	11,277	288	61	205	27
	SIKE 6-cores (Total) [17]	Virtex-7	50,084	69,054	19,892	576	55	202	32
	SIKE 3-cores (Total) [26]	Virtex-7	49,099	62,124	18,711	294	23	226	32
Lattice	NewHope KEX Server [18]	Artix-7	20,826	9,975	7,153	8	14	131	13,699
	NewHope KEX Client [18]	Artix-7	18,756	9,412	6,680	8	14	133	12,723
	NewHope KEX Server [22]	Artix-7	5,142	4,452	1,708	2	4	125	731
	NewHope KEX Client [22]	Artix-7	4,498	4,635	1,483	2	4	117	653
	FrodoKEM-640 KeyGen [12]	Artix-7	3,771	1,800	1,035	1	6	167	51
	FrodoKEM-640 Encaps [12]	Artix-7	6,745	3,528	1,855	1	11	167	51
	FrodoKEM-640 Decaps [12]	Artix-7	7,220	3,549	1,992	1	16	162	49
H	SPHINCS-256 (Total) [3]	Kintex-7	19,067	3,132	7,306	3	36	525	654
OWF	Picnic-L1 Sign [15]	Artix-7	76,472	21,061	–	–	53	125	3,994
	Picnic-L1 Verify [15]	Artix-7	68,614	16,821	–	–	34	125	4,223

Table 2: Implemented FrodoKEM parameter sets.

	Security	n	q	σ	Ciphertext Size
FrodoKEM-640	128-bit	640	2^{15}	2.8	9,720 Bytes
FrodoKEM-976	192-bit	976	2^{16}	2.3	15,744 Bytes

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 'error' value (as in Equation 1). Some smaller operations such as message encoding is also required. The ciphertexts are the output of these calculations and are used to calculate a shared secret (**ss**) via SHAKE. The matrices generated heavily utilize the randomness sources, suggested by the authors via AES or SHAKE. The output of these algorithms have nice statistical properties, but the overhead required to achieve this is high.

Naehrig et al. [19] 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 by inversion sampling using a small look-up table which corresponds to the discrete cumulative density functions (CDT sampling).

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 $\text{seed}_\mathbf{A} \leftarrow H(\mathbf{z})$
4: Generate $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ via $\mathbf{A} \leftarrow \text{Frodo.Gen}(\text{seed}_\mathbf{A})$
5: Generate $\mathbf{S} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_\mathbf{E}, n, \bar{n}, T_\chi, 1)$
6: Generate $\mathbf{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_\mathbf{E}, n, \bar{n}, T_\chi, 2)$
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**

Algorithm 2 FrodoKEM encapsulation

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

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 randomness $\text{seed}'_{\mathbf{E}}||\mathbf{k}'||\mathbf{d}' \leftarrow G(pk||\mu')$
  Generate $\mathbf{S}' \leftarrow$ Frodo.SampleMatrix($\text{seed}'_{\mathbf{E}}, \bar{m}, n, T_\chi, 4$)
  Generate $\mathbf{E}' \leftarrow$ 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 $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
  Generate $\mathbf{E}'' \leftarrow$ Frodo.SampleMatrix($\text{seed}'_{\mathbf{E}}, \bar{m}, n, T_\chi, 6$)
  Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' +$ Frodo.Encode(μ')
  if $\mathbf{B}'||\mathbf{C} = \mathbf{B}''||\mathbf{C}'$ and $\mathbf{d} = \mathbf{d}'$ **return** $\mathbf{ss} \leftarrow F(\mathbf{c}_1||\mathbf{c}_2||\mathbf{k}'||\mathbf{d})$
  else return $\mathbf{ss} \leftarrow F(\mathbf{c}_1||\mathbf{c}_2||\mathbf{s}||\mathbf{d})$
end procedure

There has been a number of software and hardware optimizations of FrodoKEM. Howe et al. [12] 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 randomness, 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 parallelize multipliers and incurs high memory costs.

So far there has been little investigation of side-channel analysis for FrodoKEM other than ensuring the implementations run in constant-time [12]. Bos et al. [8] have investigated FrodoKEM in terms of its resistance against power analysis. They find that the secret-key is recoverable for a number of different scenarios, requiring a small amount of traces (< 1000) for any of the parameter sets. They propose a simple countermeasure to twart their attack by changing the order during the inner product multiplication. A previous attack on Frodo by Aysu et al. [4] also suggests using random shuffling or by adding dummy instructions.

Thus, to counter this type of attack, it is important for masking to be investigated, and evaluated in terms of its practical performance. NIST have also stated many times that masking and countermeasures are an important evaluation criteria for analysing these post-quantum candidates [21, 2].

2.3 SHAKE as a Seed Expander

The pqm4 project nicely summarises the percentage of time each post-quantum candidate spends using SHAKE in software [16, Section 5.3]. This shows that Kyber, NewHope, Round5, Saber, and ThreeBears spend upwards of 50% of their total runtimes using SHAKE in some form or another. For signature schemes, this value can reach upwards of 70% in some cases.

There has been previous investigations of using alternatives to SHAKE in software for NIST post-quantum standardisation candidates. Bos et al. [9] recently improved the throughput of software implementations of FrodoKEM by leveraging a different randomness source for generating the matrix $\mathbf{A}$; xoshiro128**, increasing the throughput by 5x. Round5 has also been shown to improve its performance using an alternative randomness source [27], instead using a candidate from NIST's lightweight competition, which shows a performance improvement by 1.4x. SPHINCS+, using Haraka, has also been shown to have a 5x speed-up when considered instead of SHAKE [5]. These recent reports show there is room for further investigations (in hardware) for using SHAKE in post-quantum cryptographic schemes. Moreover, alternative random sources may be required for these NIST PQC schemes once they are integrated into the real-world; e.g. in a Hardware Security Module (HSM) which require randomness from physical processes, i.e. a True Random Number Generator (TRNG). Despite these investigations into alternative randomness sources, utilising sources not specified in the scheme's specifications may break compatibility, which would be the case for FrodoKEM which only considers AES and SHAKE.

2.4 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 [21]. 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 and hiding. 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 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 be applied to FrodoKEM at a very low cost. We therefore argue the overhead that a masked implementation of FrodoKEM in hardware incurs is minimal, hence making it a strong candidate when side-channel analysis is a concern. In FrodoKEM the only operation using the secret matrix $\mathbf{S}$ is the computation of the matrix $\mathbf{M}$ as $\mathbf{C} - \mathbf{B'S}$ during decapsulation. When $\mathbf{S}$ is split in two (or more) shares $\mathbf{S_i}$ using addition modulo q, the above multiplication by $\mathbf{B'}$ can be simply applied to all shares independently. Results are then subtracted by $\mathbf{C}$ one-by-one, so that computations never depend on both shares simultaneously. More precisely, in a two share scenario the flow of computation would be that first $\mathbf{B'S_1}$ is generated and subtracted from $\mathbf{C}$ to produce the first share of $\mathbf{M}$ as $\mathbf{M_1} = \mathbf{C} - \mathbf{B'S_1}$; then the actual value of $\mathbf{M}$ is derived via $\mathbf{M} = \mathbf{M_1} - \mathbf{B'S_2}$. All intermediate steps operate on values where at least one element is unknown to the adversary and thus no classical DPA style attack can succeed.

Masking can only be successful if an implementation features a low enough signal to noise ratio. Otherwise, single trace attacks, i.e. attacks where secrets can be extracted by analysing each trace individually, will succeed. Masking in itself cannot overcome this threat. Either an implementation has sufficient parallelism to ensure that the signal to noise ratio is sufficiently low, or, hiding countermeasures need to be deployed. Hiding countermeasures in hardware can take advantage of "unused" circuitry, e.g. it is possible to ensure parts of the circuit are always active; in our implementation we try to achieve this anyway to ensure high throughput. Other hiding countermeasures often increase the noise for the adversary by reordering of operations or the addition of dummy operations. E.g. in the context of matrix multiplications, one can, with minimal overhead, change the ordering of the processing of rows/-columns and even the ordering of the computation of the partial products. Such measures typically imply a small amount of extra circuitry when implemented in hardware, but they do not result in a different architecture for the matrix multiplication across different other choices.

3 Hardware Design

Our main design goal is to improve the throughput of the lattice-based key encapsulation scheme FrodoKEM [19] 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 [20], currently a semi-finalist in the process. 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 optimizations applicable to FrodoKEM has yet to be done. For instance, previous implementations do not consider parallelisations or other design alternatives capable of significantly improving the throughput.

As described in Section 2, FrodoKEM requires heavy use of randomness generation and/or seed expanding. In the algorithm specifications, it is suggested to use either SHAKE or AES. In particular, the most computationally intensive operations, such as Line 11 of Algorithm 3, require 410k or 953k 16-bit pseudo-random values, depending on the parameter set used. In order for the generation of randomness not to be the bottleneck it needs to achieve a very high throughput (ideally with relatively low area consumption) typically in the range of 16 bits per clock cycle. In a previous hardware design, proposed by Howe et al. [12], 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, potentially increasing the operating frequency of the design, and thus improve its throughput. Moreover, parallelisations were not possible for this design, as this would either require a faster SHAKE design, increasing the area consumption by 3-8x [6] or worse still having several SHAKE instances, incurring an even worse resource consumption overhead. The area consumption of SHAKE (or AES) was an issue with the previous hardware design. For example, cSHAKE used within FrodoKEM-640 Encaps occupies 42% of the overall hardware resources [12].

To improve the parallelism of our implementation, we further the discussions in Section 2.3. That is, we further the investigations that research alternative sources of randomness in post-quantum cryptographic schemes and translate this into hardware. As with other design

explorations, this means we do not completely comply with the specifications (and test vectors) by not using a NIST standard. However, their security arguments that AES is an 'ideal cipher' for use as an seed expander still apply as we replace this with Trivium, as it has analogous security properties of being indistinguishable from random. Trivium does not provide the same level of classical security as AES or SHAKE, however it is used to randomly generate a public element and suffices to eliminate the possibility of backdoors and all-for-the-price-of-one attacks [19]. Moreover, with NIST's lightweight competition happening in parallel, it is likely that there will be future NIST standards that are more efficient than SHAKE. We may also see specific use-cases where an alternative PRNG is preferred to SHAKE. Thus, considering alternative PRNGs as a design exploration is an important contribution to the standardisation process.

We explored several options for the randomness source used in the Frodo.Gen operation, that is, sampling the matrix $\mathbf{A}$, and we decided to integrate an unrolled x32 Trivium [10] implementation into our design. The use of alternative PRNG sources is discussed in the FrodoKEM specifications, specifically they state that "the distribution of matrix $\mathbf{A}$ from a truly uniform distribution to one generated from a public random seed in a pseudorandom fashion does not affect the security of FrodoKEM or FrodoPKE, provided that the pseudorandom generator is modeled either as an ideal cipher (when using AES128) or a random oracle (when using SHAKE128)", thus we use Trivium as our 'ideal cipher', which also maintains good statistical pseudo-randomness properties as well as the high throughput performances we need for our designs.

3.1 Hardware Optimisations

In order to fully explore the potential of FrodoKEM in hardware, we propose several architectures characterized by different design goals (in terms of throughput). We use the proposed architecture to implement key generation, encapsulation, and decapsulation, on two sets of parameters proposed in the specifications: FrodoKEM-640 and FrodoKEM-976. Our designs use 1x, 4x, 8x, and 16x parallel multiplications during the most computationally intensive parts in FrodoKEM. These operations are the LWE matrix multiplications of the form:

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

required in key generation, encapsulation, and decapsulation. In the previous hardware implementations of

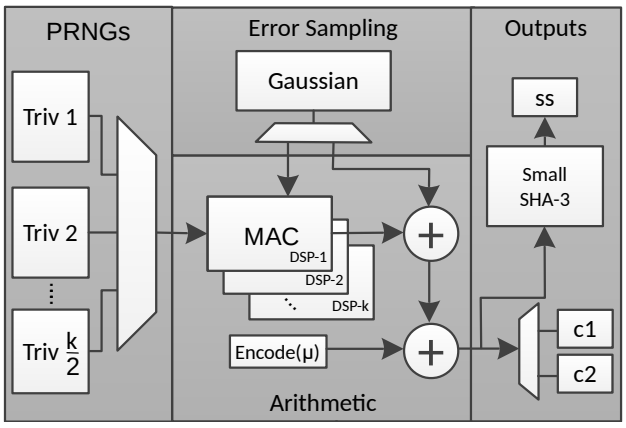

Fig. 1: A high-level overview of the proposed hardware designs for FrodoKEM for k parallel multipliers. The architecture is split into sections 'PRNGs' for Trivium modules, 'Error Sampling' for the Gaussian sampler, 'Arithmetic' for the LWE multiplier, and 'Outputs' for the shared-secret and ciphertexts.

FrodoKEM, the operations of the type of Equation 1 took approximately 97.5% of the overall run-time of the designs [12]. As in the literature, we exploit DSP slices on the FPGA for the multiply-and-accumulate (MAC) operations required for matrix multiplication. Hence, each parallel multiplication of the proposed designs requires its own DSP slice. The LWE matrix multiplication component incurs a large computational overhead. Because of this, it is an ideal target for optimizations, and for our optimizations we heavily rely on parallelisations. 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.

Figure 1 shows a high-level overview of the hardware architecture and the following descriptions will link to the design overview. The `Arithmetic` part of the LWE core is essentially made by vector-matrix multiplication (that is, $\mathbf{S}[\text{row}] \times \mathbf{A}$), addition of a `Gaussian` error value (that is, $\mathbf{E}[\text{row}, \text{col}]$), 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 (a larger) 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 run-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 outputted

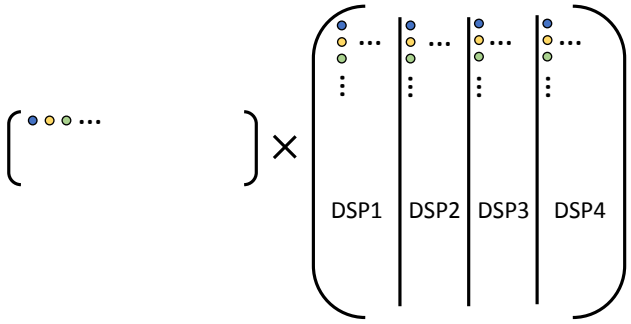

Fig. 2: Parallelising matrix multiplication, for $\mathbf{S} \times \mathbf{A}$, used within LWE computations for an example of $k = 4$ parallel multiplications, using $k = 4$ DSPs on the FPGA.

ciphertext values are also consistently added into an instantiation of SHAKE, which is required to calculate the shared secret. This process is pipe-lined to ensure high throughput and constant run-time.

To avoid using BRAM (for pre-computing some of the matrix $\mathbf{A}$) and while keeping the throughput needed by the MAC operations of the matrix multiplications, the designs require 16 bits of pseudo-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[2]. This is shown in `PRNGs` part of Figure 1. This pseudo-randomness forms the matrix $\mathbf{A}$ in Equation 1, whereas the matrix $\mathbf{S}$ and $\mathbf{E}$ require randomness taken from `Gaussian` sampler. The cumulative distribution table (CDT) sampler technique has been shown to be the most suitable one for hardware [11] and thus we use it in our designs. However, compared with previous works, we replace the use of AES as a pseudo-random input with Trivium. This ensures the same high throughput, but requires significantly less area on the FPGA.

The technique we use to parallelise Equation 1 is to vertically partition the matrix $\mathbf{A}$ into k equal sections, where k is the number of parallel multiplications, and DSPs, used. This is shown in Figure 2 for $k = 4$ parallel multiplications, utilizing 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 $\mathbf{S}$. 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 $\mathbf{A}$ is set to be a 16-bit integer and each output from Trivium is 32 bits, that is, two 16-bit integers. As the Trivium modules are relatively small in area consumption on the FPGA (169 slices), an increase in k is fairly scalable as an impact on the overall design.

3.2 Efficient First-Order Masking

We implement first-order masking scheme (discussed in Section 2.4) 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 us to implement this masking schema without affecting the area consumption or throughput. Essentially this is achieved by re-using the parallelised matrix multiplier used through the proposed hardware design for FrodoKEM. The matrix $\mathbf{S}$ is split using the same technique from Figure 2 and our secret shares are generated by using the Trivium modules as a PRNG source. By computing these calculations in parallel, the masked calculation of $\mathbf{M}$ has the same run-time as the one needed to complete the calculation when masking is not used. We ensure that the same row-column operation during the matrix multiplication is not computed in each parallel operation, to circumvent any attack that might combine the power traces and essentially remove the masking. To ensure this countermeasure operates effectively and has no implementation mistakes, one should further this by performing TVLA analysis.

3.3 A note on Software Implementations

The reason the performance of Trivium was investigated for use within FrodoKEM is due to Trivium's outstanding performance *specifically* in hardware, which was the reason it was chosen for the eSTREAM project. However, one should not expect a similar performance gain by using Trivium in FrodoKEM in software. To demonstrate, we can take the performance of the AES implementation [28] used by pqm4 [16] which operates at 101 cycles per byte on ARM Cortex M3/M4 and a Trivium implementation [1] which operates at 36 cycles per byte on ARM Cortex M0[3]. We should additionally consider that in the FrodoKEM-640 implementation using AES; key generation, encapsulation, and decapsulation respectively use AES for 73.6%, 77.1% and 76.3% of its overall clock cycles. Thus, all other

[2] For comparison, the AES implementation used in [23] generates 128 bits of randomness in 13 cycles and requires 349 slices and 2 BRAMs on the FPGA. This makes Trivium 3.25x faster than AES whilst required less hardware resources.

[3] Some differences between the ARM Cortex M0, M3, and M4 exist which may affect this comparison, such as the advanced data processing bit field manipulations on the M3 and M4, or the SIMD and fast multiply-and-accumulate on the M4.

things being equal, when replacing AES with Trivium we might see an increase of 1.9-2x in the overall runtime of FrodoKEM-640 implementation.

4 Results

In this section we present the results obtained when implementing our FrodoKEM architecture. We provide a table of results for each of the key generation, encapsulation, and decapsulation designs in Tables 3, 4, and 5, respectively. We also provide results for the PRNG and Gaussian sampler in Table 6. All tables give comparative results of the previous FrodoKEM design in hardware, which utilize 1x LWE multiplier per clock cycle and completely conform to the FrodoKEM specifications by using cSHAKE where we are using Trivium. Moreover, all results are benchmarked on the same FPGA device as previous work, Xilinx Artix-7 XC7A35T FPGA, running on Vivado 2019.1.

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 area on the FPGA (measured in 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 parallelising, is essentially the only reason for area increase when we move from a base design 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. There is obviously an increase when we move from parameter sets due to the matrix $\mathbf{A}$ increasing from 640 to 976 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 utilize 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 utilized. Figure 3 shows a slice count summary of all the proposed designs, showing a consistent and fairly linear increase in slice utilization 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 (i.e., they overlap). In most cases slice counts at least double for decapsulation when BRAM is removed, with only slight increases in throughout, hence it might be not be useful in some use cases. BRAM usage, however, is not as friendly when hardware designs are considered for ASIC, thus is it useful to consider designs both with and without BRAM.

By changing our source of randomness and parallelising the most computationally heaving components in FrodoKEM we have shown significant improvements in FPGA area consumption and throughput performance compared to the previous works. For instance, comparing to FrodoKEM module [12] (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 utilize BRAM. For decapsulation, we decrease the amount of slices used between 1.6x and 2.6x when BRAM is used and similarly decrease slice counts by 1.5x and 1.1x when BRAM is not used. These savings are expected since more than half of this is due to storage otherwise used in BRAM.

Tables 3, 4, and 5 also contain a metric to analyse the area-time efficiency of the proposed designs. This metric takes into account the hardware design's utilisation of slices (i.e., not BRAM) and the time taken (in this case, seconds) for a full key generation, encapsulation, or decapsulation operation to complete. There is a common trend when analysing these results; the hardware designs become significantly more performant when the number of parallel multipliers are increase.

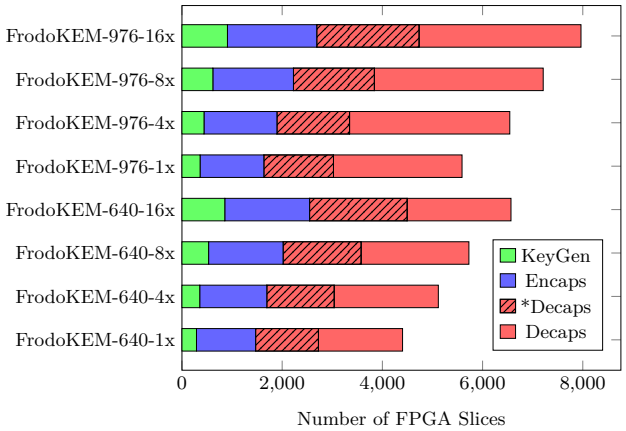

Fig. 3: Visualisation of FPGA slice consumption of FrodoKEM's key generation, encaps, and decaps on a Xilinx Artix-7. Decaps values overlap to show results with (*) and without BRAM.

Table 3: FPGA resource consumption of the proposed FrodoKEM **KeyGen** designs, using 1, 4, 8, and 16 parallel multipliers, for both parameter sets, on a Xilinx Artix-7 FPGA.

FrodoKEM Protocol	LUT	FF	Slices	DSP/ BRAM	MHz	Ops/ Sec	Area×Time (Slices×Secs)
KeyGen-640 1x	971	433	290	1/0	191	59	4.92
KeyGen-640 4x	1,174	781	355	4/0	185	226	1.58
KeyGen-640 8x	1,679	1,570	532	8/0	182	445	1.20
KeyGen-640 16x	2,587	2,994	855	16/0	172	840	1.02
KeyGen-640 [12]	3,771	1,800	1,035	1/6	167	51	20.29
KeyGen-976 1x	1,243	441	362	1/0	189	25	14.48
KeyGen-976 4x	1,458	792	440	4/0	184	97	4.54
KeyGen-976 8x	1,967	1,576	617	8/0	178	187	3.30
KeyGen-976 16x	2,869	3,000	908	16/0	169	355	2.56
KeyGen-976 [12]	7,139	1,800	1,939	1/8	167	22	88.14

Table 4: FPGA resource consumption of the proposed FrodoKEM **Encapsulation** designs, using 1, 4, 8, and 16 parallel multipliers, for both parameter sets, on a Xilinx Artix-7 FPGA.

FrodoKEM Protocol	LUT	FF	Slices	DSP/ BRAM	MHz	Ops/ Sec	Area×Time (Slices×Secs)
Encaps-640 1x	4,246	2,131	1,180	1/0	190	58	20.34
Encaps-640 4x	4,620	2,552	1,338	4/0	183	221	6.05
Encaps-640 8x	5,155	3,356	1,485	8/0	177	427	3.48
Encaps-640 16x	5,796	4,694	1,692	16/0	171	825	2.05
Encaps-640 [12]	6,745	3,528	1,855	1/11	167	51	36.37
Encaps-976 1x	4,650	2,118	1,272	1/0	187	25	50.88
Encaps-976 4x	4,996	2,611	1,455	4/0	180	94	15.47
Encaps-976 8x	5,562	3,349	1,608	8/0	175	183	8.79
Encaps-976 16x	6,188	4,678	1,782	16/0	168	350	5.09
Encaps-976 [12]	7,209	3,537	1,985	1/16	167	22	90.22

Table 5: FPGA resource consumption of the proposed FrodoKEM **Decapsulation** designs, using 1, 4, 8, and 16 parallel multipliers, for both parameter sets, on a Xilinx Artix-7 FPGA. Asterisk (*) denotes designs that used BRAM.

FrodoKEM Protocol	LUT	FF	Slices	DSP/ BRAM	MHz	Ops/ Sec	Area×Time (Slices×Secs)
Decaps-640 1x	10,518	2,299	2,933	1/0	190	57	51.46
Decaps-640 4x	11,581	2,818	3,424	4/0	174	208	16.46
Decaps-640 8x	13,128	3,737	3,710	8/0	164	391	9.49
Decaps-640 16x	14,528	5,335	4,020	16/0	160	763	5.27
*Decaps-640 1x	4,466	2,152	1,254	1/12.5	162	49	25.59
*Decaps-640 4x	4,841	2,661	1,345	4/12.5	161	192	7.00
*Decaps-640 8x	5,476	3,479	1,558	8/12.5	156	372	4.19
*Decaps-640 16x	6,881	5,081	1,947	16/12.5	149	710	2.74
Decaps-640 [12]	7,220	3,549	1,992	1/16	162	49	40.65
Decaps-976 1x	14,217	2,295	3,956	1/0	188	25	158.24
Decaps-976 4x	16,234	2,853	4,648	4/0	170	88	52.82
Decaps-976 8x	17,451	3,687	4,985	8/0	161	167	29.85
Decaps-976 16x	18,960	5,285	5,274	16/0	157	325	16.23
*Decaps-976 1x	4,888	2,153	1,390	1/19	162	21	66.19
*Decaps-976 4x	5,259	2,662	1,450	4/19	160	83	17.47
*Decaps-976 8x	5,888	3,490	1,615	8/19	155	161	10.03
*Decaps-976 16x	7,213	5,087	2,042	16/19	148	306	6.67
Decaps-976 [12]	7,773	3,559	2,158	1/24	162	21	102.76

Table 6: FPGA resource consumption of the proposed PRNG and Error Sampler designs on a Xilinx Artix-7 FPGA.

FrodoKEM Protocol	LUT	FF	Slices	DSP/ BRAM	MHz	Ops/ Sec
Error+Trivium	401	311	179	0/0	211	211m
Trivium	296	299	169	0/0	220	220m
Error+AES [12]	1,901	1,140	756	0/0	184	184m
cSHAKE [12]	2,744	1,685	766	0/0	172	1m

We also see this in the increase in throughput performance in Figure 4. Moreover, we can use this metric to compare with previous work by Howe et al. [12] to see that in all cases, parallelising results provide significant speed-ups; up to 35x improvement for key generation.

Since the majority of our proposed designs operate without BRAM[4], we are able to attain a higher frequency than previous works. Overall our throughput outperforms previous *comparable* results, by factors between 1.13x and 1.19x [12]. Moreover, whilst maintaining less area consumption than previous research, we are able to increase the amount of parallel multipliers. 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). We also maintain the constant run-time which the previous implementation attains, as well as implementing first-order masking during decapsulation[5]. The masking is also done using parallel multiplication and thus does not affect the run-time of the decapsulation module. The clock cycle counts for each module are easy to calculate; key generation requires $(n^2\bar{n})/k$ clocks, encapsulation requires $(n^2\bar{n}+\bar{n}^2n)/k$ clocks, and decapsulation requires $(n^2\bar{n}+2\bar{n}^2n)/k$ clocks, for dimensions $n = 640$ or 976, $\bar{n} = 8$, and k referring to the number of parallel multipliers used.

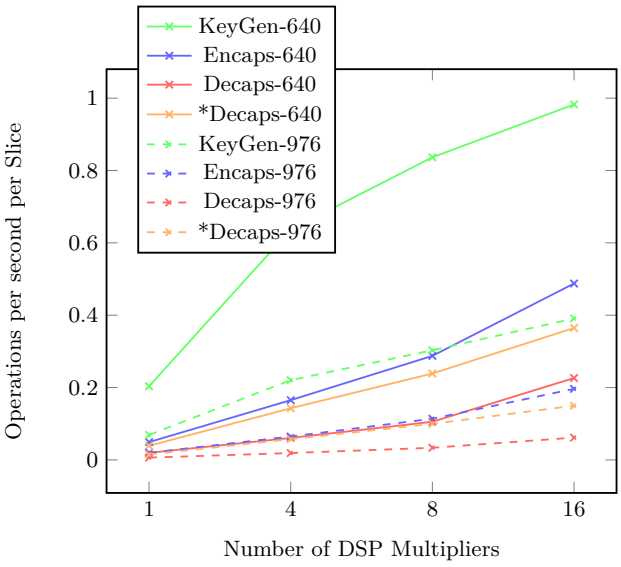

Fig. 4: Comparison of the throughput performance per FPGA slice on a Xilinx Artix-7.

5 Conclusions

The main contributions of this research is to evaluate the performance potential of FrodoKEM [19], a NIST post-quantum candidate for key encapsulation, when utilising a significantly more performant PRNG in hardware. We develop designs which can reach up to 825 operations per second, where most of the designs fit in under 1500 slices. Area consumption results are less than the previous state-of-the-art, and are much lower than many of the other post-quantum hardware designs shown in Table 1. We significantly improve the throughput performance compared to 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 an inefficient PRNG previously used, cSHAKE, with a much faster and smaller PRNG, Trivium. As a result, we are able to obtain either a much lower FPGA footprint (up to 5x smaller) or a much higher throughput (up to 16x faster) compared to previous research. Our implementations 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. We expect this research would have an impact on real-world use cases such as in TLS, as shown previously from key exchange version of Frodo [7], potentially making its performance competitive with classical cryptographic schemes used today.

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 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.

[4] We ensure BRAM is not inferred in our designs by setting `-max_bram` to zero for synthesis in Vivado.

[5] This masking could also be used in key generation (Line 7 of Algorithm 1), however the hardware results provided only show this for decapsulation.

Acknowledgements

This research was partially funded by the Innovate UK project 105747 (Hardware assisted post-quantum cryptography for embedded system devices), the EPSRC via grant EP/N011635/1 (LADA), and the ERC via grant 725042 (SEAL).

References

1. Aerabi, E., Bohlouli, M., Livany, M.H.A., Fazeli, M., Papadimitriou, A., Hely, D.: Design space exploration for ultra-low-energy and secure iot mcus. ACM Transactions on Embedded Computing Systems (TECS) **19**(3), 1–34 (2020)
2. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Kelsey, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., et al.: Status report on the second round of the NIST post-quantum cryptography standardization process. NIST, Tech. Rep., July (2020)
3. Amiet, D., Curiger, A., Zbinden, P.: FPGA-based Accelerator for Post-Quantum Signature Scheme SPHINCS-256. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 18–39 (2018)
4. Aysu, A., Tobah, Y., Tiwari, M., Gerstlauer, A., Orshansky, M.: Horizontal side-channel vulnerabilities of post-quantum key exchange protocols. In: 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 81–88. IEEE (2018)
5. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS+ Signature Framework. Cryptology ePrint Archive, Report 2019/1086 (2019), https://eprint.iacr.org/2019/1086
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Keccak implementation overview. URL: http://keccak. neokeon. org/Keccak-implementation-3.2. pdf (2012)
7. 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)
8. Bos, J.W., Friedberger, S., Martinoli, M., Oswald, E., Stam, M.: Assessing the Feasibility of Single Trace Power Analysis of Frodo. In: International Conference on Selected Areas in Cryptography. pp. 216–234. Springer (2018)
9. 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
10. De Canniere, C., Preneel, B.: Trivium. In: New Stream Cipher Designs, pp. 244–266. Springer (2008)
11. Howe, J., Khalid, A., Rafferty, C., Regazzoni, F., O'Neill, M.: On practical discrete Gaussian samplers for lattice-based cryptography. IEEE Transactions on Computers **67**(3), 322–334 (2018)
12. 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)
13. Information technology — Security techniques — Lightweight cryptography — Part 3: Stream ciphers. Standard, International Organization for Standardization, Geneva, CH (2012)
14. Jang, J.w., Choi, S., Prasanna, V.: Area and time efficient implementations of matrix multiplication on fpgas. In: 2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings. pp. 93–100. IEEE (2002)
15. Kales, D., Ramacher, S., Rechberger, C., Walch, R., Werner, M.: Efficient FPGA Implementations of LowMC and Picnic. In: Cryptographers' Track at the RSA Conference. pp. 417–441. Springer (2020)
16. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. NIST's Second PQC Standardization Conference (2019), https://eprint.iacr.org/2019/844
17. Koziel, B., Azarderakhsh, R., Kermani, M.M.: A high-performance and scalable hardware architecture for isogeny-based cryptography. IEEE Transactions on Computers **67**(11), 1594–1609 (2018)
18. Kuo, P.C., Li, W.D., Chen, Y.W., Hsu, Y.C., Peng, B.Y., Cheng, C.M., Yang, B.Y.: High performance post-quantum key exchange on FPGAs. Cryptology ePrint Archive, Report 2017/690 (2017), https://eprint.iacr.org/2017/690
19. 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
20. NIST: Post-quantum crypto project. http://csrc.nist.gov/groups/ST/post-quantum-crypto/ (2016)
21. 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)
22. Oder, T., Güneysu, T.: Implementing the NewHope-simple key exchange on low-cost FPGAs. Progress in Cryptology–LATINCRYPT **2017** (2017)
23. Pöppelmann, T., Güneysu, T.: Towards practical lattice-based public-key encryption on reconfigurable hardware. In: International Conference on Selected Areas in Cryptography. pp. 68–85. Springer (2013)
24. Qasim, S.M., Abbasi, S.A., Almashary, B.: A proposed fpga-based parallel architecture for matrix multiplication. In: APCCAS 2008-2008 IEEE Asia Pacific Conference on Circuits and Systems. pp. 1763–1766. IEEE (2008)
25. 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
26. Roy, D.B., Mukhopadhyay, D.: Post Quantum ECC on FPGA Platform. Cryptology ePrint Archive, Report 2019/568 (2019), https://eprint.iacr.org/2019/568
27. Saarinen, M.J.O.: Exploring NIST LWC/PQC Synergy with R5Sneik: How SNEIK 1.1 Algorithms were Designed to Support Round5. Cryptology ePrint Archive, Report 2019/685 (2019), https://eprint.iacr.org/2019/685
28. Schwabe, P., Stoffelen, K.: All the aes you need on cortex-m3 and m4. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptology – SAC 2016. Lecture Notes in Computer Science, vol. 10532, pp. 180–194. Springer-Verlag Berlin Heidelberg (2017), document ID: 9fc0b970660e40c264e50ca389dacd49, https://cryptojedi.org/papers/#aesarm

29. 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)
30. Wang, W., Szefer, J., Niederhagen, R.: FPGA-based Niederreiter cryptosystem using binary Goppa codes. In: International Conference on Post-Quantum Cryptography. pp. 77–98. Springer (2018)