Fast Quantum-Safe Cryptography on IBM Z

Jonathan Bradbury[1] and Basil Hess[2]

[1] IBM Systems, Poughkeepsie, USA
[2] IBM Research Europe, Rueschlikon, Switzerland

Abstract. Performance of software implementations on today's available hardware architectures plays a crucial role in the adoption of quantum-safe cryptography. An important target for quantum-safety are IBM Z® systems, which run and secure a majority of all worldwide transactions. With its current z15 architecture, the platform offers a range of ISA extensions suitable for optimizing quantum-safe algorithms. In this work, we present optimizations of two promising candidates in the third round of the NIST PQC standardization process: SIKE and Dilithium. Our SIKE implementation covers NIST security levels 1-5. It uses vectorization techniques for its $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic and achieves a significant speedup compared to generic implementations, running in 3.4 ms (encaps + decaps) for NIST level 1. Our Dilithium implementation benefits from vector optimizations applied to NTT and to sampling, and from SHA3 instructions on z15, running in 42.8 µs (sign) and 14.7 µs (verify) for NIST level 2. We present insights on the z15 ISA, on the implementations, evaluation results and provide an outlook of further optimization potential.

Keywords: Quantum Safe, IBM Z, SIKE, Dilithium, Optimization, Evaluation

1 Introduction

NIST is in the third round of the standardization process for post-quantum cryptographic schemes to address the upcoming threat that quantum computers pose on most of the public-key cryptography used today. While security against quantum attacks is the primary motivation for the standardization, there are several factors that will play an important role for a timely adoption in practice. The industry is already preparing the migration to promising candidates for standardization, where considerations such as security, bandwidth, performance and applicability to the underlying infrastructure and platform are central.

An example of such a platform are IBM Z systems. They are in use by a majority of the largest companies and are responsible to handle a vast number of all worldwide transactions [11]. To evaluate the performance potential on today's available z15 CPU architecture, we select two promising candidates from the third round: SIKE and Diltithium. We give an overview of useful ISA extensions of the z15 architecture and demonstrate how we apply optimizations like vectorization and SHA3 acceleration to the underlying algorithms. Our evaluation results show that the algorithms get significant speedups by using the ISA extensions and they present z15 as an attractive target for optimizations.

1.1 Optimized candidates

We selected two candidates with different interesting characteristics: Superingular Isogeny Key Encapsulation (SIKE) [3] and CRYSTALS-Dilithium (or short Dilithium) [6].

SIKE is a KEM based on isogenies of supersingular elliptic curves. The underlying key agreement scheme [4] Supersingular Isogeny Diffie Hellman (SIDH) can be seen as closely related to classical schemes like ECDH and DH, whereas still providing security against quantum attacks. An interesting property is that SIKE has the smallest key and ciphertext sizes among all NIST candidates, making it attractive for bandwidth or storage constrained settings. The main drawback also noted in the NIST report on the second round candidates [2] is its relatively slow performance compared to other candidates. More optimized implementation are therefore of high importance for SIKE. There are already several optimized implementations available for SIKE, for instance for Intel x64 [3], ARMv8-A [10], ARMv7-M [9] and initial work for AVX512 [8]. We were especially interested in how the z15 vector instructions can be efficiently applied to accelerate SIKE.

Dilithium is a signature scheme based on the module learning with errors problem (MLWE). It one of two lattice-based signature schemes in the third round (the other one is Falcon). NIST [2] sees Dilithium as well balanced in terms of performance and key/signature sizes, and it is further considered to allow a simpler implementation compared to Falcon. The availability of Dilithium variants using AES further makes it attractive for targets with AES acceleration. Since the z15 architecture offers SHA3 instructions, we were especially interested how this accelerates the default Dilithium variants that use SHA3/SHAKE as well as how it compares to the hardware accelerated AES on the z15.

2 IBM z15

IBM z15 was released in 2019 and is a 64-bit big-endian multi-core architecture with roots going back to the System/360 from 1964. An IBM Z system consists of many cores with 5.2 GHz clock frequency each. Of especial interest for us are the *Vector Facility*, *Vector-Enhancements Facility 1/2* with 32 128-bit vector registers and the *Message-Security-Assist Extension 6* that offer SHA3 and SHAKE instructions. We discuss a few instructions in more detail, a comprehensive list is available in [1].

Vector Multiply Sum Logical - VMSL VMSL combines two 56-bit multiplications and one full 128-bit addition in a single SIMD instruction. The inputs are: (i) two vectors each with one 56-bit value stored in the lower and one in the upper part, respectively: $a = (a_0, a_1)$ and $b = (b_0, b_1)$. (ii) one vector with a full 128-bit input acting as an accumulator: c. (iii) a value $d \in \{0, 4, 8, 12\}$. We say that $d_0 = 2$ if $d \in \{4, 12\}$, else $d_0 = 1$; and $d_1 = 2$ if $d \in \{8, 12\}$, else $d_1 = 1$. The result is a 128-bit value $e = (a_0 b_0 d_0) + (a_1 b_1 d_1) + c$. This design allows at least 2^{14} VMSL instructions to be chained together without an overflow. The VMSL instruction was introduced on the IBM z14 with the *vector-enhancements facility 1*.

Vector Add - VA, VAC, VACC, VACCC Vector Add instructions perform an unsigned addition of two 128-bit words (or multiple 64, 32, 16, or 8-bit values within a 128-bit vector). There are several variants with carry-in (VAC, VACCC), carry-out (VACC, VACCC) and without carry (VA). The vector add instructions were introduced with the *vector facility* on IBM z13.

Vector Subtract - VS, VSCBI, VSBI, VSBCBI Vector Subtract instructions perform an unsigned subtraction of two 128-bit words (or multiple 64, 32, 16, or 8-bit values within a 128-bit vector). There are several variants with borrow-in (VSBI, VSBCBI), borrow-out (VSCBI, VSBCBI) and without borrow (VS). The vector subtract instructions were introduced with the *vector facility* on IBM z13.

SHAKE and SHA3 - KIMD, KLMD The Compute Intermediate Message Digest and Compute Last Message Digest instructions provide acceleration for hash functions. The *Message-Security-Assist Extension 6* added functions for accelerating SHAKE-128, SHAKE-256, SHA-224, SHA3-256, SHA3-384 and SHA3-512.

Further instructions Further instructions used in our implementations include vector compare equal (VCEQ), vector multiply (VML), vector multiply high (VMH), vector permute (VPERM), vector bit permute

(VBPERM), vector shift left double by byte (VSLDB), vector merge (VMRL), vector select (VSEL), and with complement (ANDC), loading and storing functions.

3 Optimizing SIKE

The arithmetic in SIKE involves operations over elliptic curves over prime fields. The primes are of the form $p = 2^{e_2}3^{e_3} -$~~1.~~ There are four parameter sets, each involving a different prime: `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`. Compared to commonly known elliptic curve cryptography, most of the arithmetic is performed in a quadratic extension field of $\mathbb{F}_p$, which we denote as $\mathbb{F}_{p^2}$. The optimizations we tackle are on the level of the $\mathbb{F}_p$ and the $\mathbb{F}_{p^2}$ arithmetic.

Our z15 implementation is based on the optimized code base from the SIKE submission (SIDH v3.3)[1]. The routines implemented in the original code use general purpose registers. Depending on the context in our optimizations, it is most efficient to keep the output of the routines in vector registers, as well as already providing inputs in vector registers. We denote variants returning vectors with the suffix `_OVEC`, variants with inputs provided in vectors with the suffix `_IVEC` and variants with inputs and outputs in vector with the suffix `_VEC`.

Modern CPU architectures such as IBM z15 have a multi-stage pipeline, where instructions are processed in a certain number of CPU cycles. A common technique to optimize pipeline use and throughput is to issue several independent instructions without data dependency. If a function has inherent data dependencies, it can make sense to interleave multiple instantiations of the function. If we interleave a function `X` ways, we identify this with a prefix `XWAY_` in the function name. In the following, we describe the optimized routines in more detail.

Digit Order Multi-precision integers used for SIKE are internally stored in arrays of 64-bit digits. Our implementation uses conversions from 64-bit general purpose registers to 128-bit wide vector registers. Since VMSL operates on pairs of 56-bit integers, it is most efficient to be able to read and store multi-precision integers in increments of 56 bits. While the SIKE library internally stores digits in little-endian order, we adapted the library with the possibility to reverse this to an internal big-endian order to suit the IBM Z architecture.

Limbification To be able to use VMSL efficiently, inputs originally in radix-2^{64} need to be converted to radix-2^{56}. We also call a digit in radix-2^{56} a limb. For the purpose of the conversion, we first load the full word in increments of 128 bits to vector registers (using VL), followed by shifting with VSLD and vector permute (VPERM), obtaining vectors containing a pair of radix-2^{56} limbs each.

Radix-2^{56} normalization and delimbification After applying VMSL on the limbs, and potential further additions, the vector registers hold values larger than 56-bits. Before applying a further VMSL operation, or before a conversion back to radix-2^{64}, the limbs need to be normalized to 56-bits each. This involves a carry propagation: In a loop, the least significant limb is shifted by 56 bits (VSLD), which is then added to the more significant limb (VA). The less significant limb is then normalized by selecting the 56 least significant bits (with an AND mask using ANDC).

We use the term delimbification to mean the conversion from 56-bit limbs to 64-bit digits. For this purpose, we simply store each 56-bit limb to an output array in adjacent order. We further implemented functions that interleave two normalization and delimbification operations for an efficient application in the $\mathbb{F}_{p^2}$ arithmetic. This improves pipeline utilization and reduces the latency due to carry propagation. We name the normalization function `NORM` and the delimbification function `DELIMB`.

[1] https://github.com/microsoft/PQCrypto-SIDH

3.1 Arithmetic in $\mathbb{F}_p$

Addition and subtraction Given $a < 2p$ and $b < 2p$, we compute $a + b \pmod{2p}$ and $a -\cancel{b} \pmod{2p}$, respectively. The z15 vector instructions allow to perform full 128-bit addition (VA, VAC) and subtraction (VS, VSBI). Carry and borrow propagation is done with VACC, VACCC, VSCBI, and VSBCBI. The reduction modulo $2p$ is performed unconditionally, while the final result is selected in constant time (with VSL) depending on a final overflow/underflow. The 32 vector registers in z15 allow all operands to be held in registers simultaneously. We also implemented a 2-way interleaved addition and subtraction to further reduce the latency impact induced by carry propagation. We name the addition functions `FPADD` and `MPADD`. The subtraction functions are named `FPSUB` and `MPSUB`.

Multi-precision multiplication Multi-precision multiplications constitute a substantial cost of the arithmetic in $\mathbb{F}_p$. Since current CPU architectures are limited to 64-bit word lengths, a multi-precision algorithm usually with quadratic complexity on the number of words needs to be applied. There are typically three approaches: (i) Schoolbook multiplication adds partial products row-wise. (ii) The Comba method is similar to schoolbook multiplication, but adds the partial products column-wise. (iii) Karatsuba's method breaks down a multiplication to the sum of two or more multiplications of smaller size. Karatsuba has the lowest number of total operations and is used by the Intel x64-optimized version of [3].

Our implementation is similar to the z14 optimized P-256 elliptic curve implementation [12] available in Golang but with larger primes. We found Comba's method the best suited to exploit SIMD parallelism: we aim to issue up to 2 times 8 independent VMSL instructions to keep the pipeline well utilized. If latency allows it, we further provide the result of a previous VMSL operation to VMSL as accumulator. If not, we prefer to issue VMSL operations with zero as accumulator and sum up the partial products later with VA. Using Comba's method, only one latency-critical normalization step is needed, compared to Karatsuba which requires two normalization steps. We name the multi-precision multiplication `MPMUL` and `FPMUL`, respectively, if it is combined with modular reduction.

Multi-precision squaring Compared to multiplication, squaring allows further optimization with VMSL since many of the partial products from the multiplications appear twice. We exploit this by using VMSL's fourth argument d that allows to selectively multiply the 56-bit multiplications by 2 before adding the accumulator, thus saving almost half of the VMSL instructions compared to multi-precision multiplication. We name the multi-precision multiplication `MPSQR` and `FPSQR`, respectively, if it is combined with modular reduction.

Montgomery reduction Multiplication in $\mathbb{F}_p$ involves a multi-precision multiplication followed by a modular reduction. The SIKE primes have special form that can be exploited to more efficiently implement Montgomery reduction. We use the observation described in [7] that define λ-Montgomery-friendly primes as $p \equiv\leftarrow -1 \pmod{2^{\lambda \cdot w}}$, where in our implementation $w = 56$ and λ is a positive integer. For the SIKE primes we have $\lambda_{p434} = 3$, $\lambda_{p503} = 4$, $\lambda_{p610} = 5$, $\lambda_{p751} = 6$. The reduction algorithm involves multiplications by $p + 1$, where the λ least significant 56-bit digits of $p + 1$ are zero, therefore reducing the number of VMSL instructions needed. We efficiently implemented the reduction algorithm using VMSL and VA instructions. In contrast to multi-precision multiplication, reduction requires one intermediary pass of normalization, thus requiring careful scheduling of instructions to avoid latency bottlenecks. For the smallest prime $p434$ we further found it beneficial to interleave two reductions which is useful for optimizing $\mathbb{F}_{p^2}$ arithmetic. We name the reduction function `REDC`.

3.2 Arithmetic in $\mathbb{F}_{p^2}$

A majority of the arithmetic in SIKE is done in the quadratic extension field of $\mathbb{F}_p$, where the extension field is defined as $F_{p^2} = \mathbb{F}_p(i)$ and $i^2 + 1 = 0$. Multiplication and squaring in F_{p^2} have the highest computational cost.

Multiplication Multiplication in $\mathbb{F}_{p^2}$ is defined as $a \cdot b = (a_0 + a_1 \cdot i) \cdot (b_0 + b_1 \cdot i)$. Rewritten as $a \cdot b = (a_0b_0 - a_1b_1) + (a_0b_1 + a_1b_0) \cdot i$, a simple implementation uses four multiplications, one addition and one subtraction in $\mathbb{F}_p$. A more optimal implementation performs the subtraction $a_0b_0 - a_1b_1$ and the addition $a_0b_1 + a_1b_0$ in limbs in radix-2^{56}, without carry propagation and before performing the normalization and Montgomery reduction. Compared to the simple version, this requires only two instead of four Montgomery reductions.

We found an implementation rewriting the formula as follows to be slightly faster: $a \cdot b = (a_0b_0 - a_1b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) - a_0b_0 - a_1b_1) \cdot i$. This requires three multiplications, two additions and three subtractions. We perform the initial two additions ($a_0 + a_1$ and $b_0 + b_1$) using the 128-bit addition as described in Sec. 3.1. The three subtractions are performed in limbs after the multiplications (and before normalization and delimbification). This operation can be fully pipelined without any carry propagation dependencies. We further have to consider that the subtraction $a_0b_0 - a_1b_1$ may underflow, which we avoid by unconditionally adding $p \cdot 2^{\lceil log_2(p)/64 \rceil \cdot 64}$ using one carry-less addition. Finally, we only use two Montgomery reduction steps followed by delimbification to obtain the final result.

Algorithm 1: Multiplication in $\mathbb{F}_{p^2}$

```
1:  function FP2MUL($a = (a_0, a_1), b = (b_0, b_1)$)
2:      $vec_{pp}$    $p \cdot 2^{\lceil log_2(p)/64 \rceil \cdot 64}$
3:      $t_0, t_1$    2WAY_FPADD($a_0, a_1, b_0, b_1$)
4:      $vec_{(a_0+a_1)\cdot(b_0+b_1)}$ = MPMUL_OVEC($t_0, t_1$)
5:      $vec_{a_0b_0}$    MPMUL_OVEC($a_0, b_0$)
6:      $vec_{a_1b_1}$    MPMUL_OVEC($a_1, b_1$)
7:      $vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0}$    MPSUB_VEC($vec_{(a_0+a_1)\cdot(b_0+b_1)}, vec_{a_0b_0}$)
8:      $vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0-a_1b_1}$    MPSUB_VEC($vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0}, vec_{a_1b_1}$)
9:      $vec_{a_0b_0-a_1b_1}$    MPSUB_VEC($vec_{a_0b_0}, vec_{a_1b_1}$)
10:     $vec_{a_0b_0-a_1b_1}$    MPADD_VEC($vec_{a_0b_0-a_1b_1}, vec_{pp}$)
11:     2WAY_NORM_VEC($vec_{a_0b_0-a_1b_1}, vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0-a_1b_1}$)
12:     2WAY_REDC_VEC($vec_{a_0b_0-a_1b_1}, vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0-a_1b_1}$)
13:     $c_0, c_1$ = 2WAY_DELIMB($vec_{a_0b_0-a_1b_1}, vec_{(a_0+a_1)\cdot(b_0+b_1)-a_0b_0-a_1b_1}$)
14:     return $c = (c_0, c_1)$
15: end function
```

Algorithm 1 achieves a speedup factor from 1.37 to 1.59 compared to the simple version. The larger parameter sets benefit most due to the larger amount of work that is done purely in vectors before any normalization and delimbification.

Squaring Squaring in $\mathbb{F}_{p^2}$ is defined as $a^2 = (a_0 + a_1 \cdot i)^2$. The first option to implement squaring is by rewriting the formula as $a^2 = (a_0^2 - a_1^2) + (2a_0a_1) \cdot i$, requiring two squarings, one multiplication, one subtraction and one addition. An alternative is to trade two squarings for one multiplication, by rewriting the formula to $a^2 = (a_0 + a_1)(a_0 - a_1) + (2a_0a_1) \cdot i$. This requires two multiplications, two additions and one subtraction in $\mathbb{F}_p$. A simple implementation would chose the second formula with the lower number of multiplications/squarings and perform all individual operations in $\mathbb{F}_p$. In a more optimized variant of the same formula, we interleave the initial three additions and subtractions $a_0 + a_0$, $a_0 + a_1$ and $a_0 - a_1$ (with a function 3WAY_ADDADDSUB) to achieve a higher throughput. This is illustrated in Algorithm 2.

In an optimized version of the first formula, we benefit from the lower computational cost from multi-precision squaring compared to multiplication. We are further able to perform the addition and subtraction

Algorithm 2: Squaring in $\mathbb{F}_{p^2}$, optimized variant 1

```
1:  function FP2SQR($a = (a_0, a_1)$)
2:      $2a_0, a_0 + a_1, a_0 - a_1$    3WAY_ADDADDSUB($a_0, a_1$)
3:      $c_1$    FPMUL($2a_0, a_1$)
4:      $c_0$    FPMUL($a_0 + a_1, a_0 - a_1$)
5:      return $c = (c_0, c_1)$
6:  end function
```

in limbs before normalizing the results of the squarings and multiplications, which maximizes throughput and avoids carry propagation dependencies. Only two Montgomery reduction steps are needed this way. This variant is illustrated in Algorithm 3.

Algorithm 3: Squaring in $\mathbb{F}_{p^2}$, optimized variant 2

```
1:  function FP2SQR($a = (a_0, a_1)$)
2:      $vec_{pp}$    $p \cdot 2^{\lceil log_2(p)/64 \rceil \cdot 64}$
3:      $vec_{a_0^2}$    MPSQR_OVEC($a_0$)
4:      $vec_{a_1^2}$    MPSQR_OVEC($a_1$)
5:      $vec_{a_0 a_1}$    MPMUL_OVEC($a_0, a_1$)
6:      $vec_{a_0^2 - a_1^2}$    MPSUB_VEC($vec_{a_0^2}, vec_{a_1^2}$)
7:      $vec_{2a_0 a_1}$    MPADD_VEC($vec_{a_0 a_1}, vec_{a_0 a_1}$)
8:      2WAY_NORM_VEC($vec_{a_0^2 - a_1^2}, vec_{2a_0 a_1}$)
9:      2WAY_REDC_VEC($vec_{a_0^2 - a_1^2}, vec_{2a_0 a_1}$)
10:     $c_0, c_1 =$ 2WAY_DELIMB($vec_{a_0^2 - a_1^2}, vec_{2a_0 a_1}$)
11:     return $c = (c_0, c_1)$
12: end function
```

We found that the performance and the choice of variant 1 and 2 depends on the parameter set. In `SIKEp434`, Algorithm 2 performs best while in `SIKEp503`, `SIKEp610` and `SIKEp751`, the performance achieved with Algorithm 3 was faster. Compared to the simple version, the optimized squaring achieves a speedup factor between 1.15 and 1.22 across the parameter sets. The speedup is lower compared to multiplication in $\mathbb{F}_{p^2}$ due to the lower number of total operations involved in squaring.

3.3 Implementation and Results

We implemented optimized versions of the SIKE parameter sets `SIKEp434`, `SIKEp503`, `SIKEp610`, and `SIKEp751`. The library is based on the official SIKE optimized third round submission to NIST [3]. For evaluation, we implemented a baseline and an optimized version:

1. A baseline portable C implementation based on the optimized, pure C version of the SIKE library. All loops were unrolled.

2. A z15 vectorized assembly version with the optimization techniques applied using assembly compiler-intrinsics. All optimized arithmetic is written in constant-time. The implementation is integrated with the SIKE optimized library.

The code was compiled on Linux on Z (Ubuntu 21.04 Beta) using clang-12, with `-march=z15 -mzvector -mvx` and `-O3` compiler flags activated. The evaluation machine was an IBM z15 logical partition (LPAR) operating at 5.2 GHz.

Overall results The overall performance consisting of encapsulation plus decapsulation takes between 3.4 ms for `SIKEp434` and 14.5 ms for `SIKEp751`. Key generation performance ranges from 1.0 ms for `SIKEp434` and 4.2 ms for `SIKEp751`. The figures are depicted in Table 1.

Table 1: Performance (in thousands of cycles) of SIKE on an IBM z15 LPAR at 5.2 GHz. Cycle counts are rounded to the nearest 10^3 cycles.

Scheme	`KeyGen`	`Encaps`	`Decaps`	**total** (`Encaps + Decaps`)
SIKEp434				
Portable C	22'771	36'807	39'089	75'897
This work	5'233 (1.01 ms)	8'676 (1.67 ms)	9'141 (1.76 ms)	17'818 (3.43 ms)
Speedup	4.4 x	4.2 x	4.3 x	4.3 x
SIKEp503				
Portable C	34'442	57'364	60'663	118'028
This work	8'200 (1.58 ms)	13'915 (2.68 ms)	14'763 (2.84 ms)	28'667 (5.51 ms)
Speedup	4.2 x	4.1 x	4.1 x	4.1 x
SIKEp610				
Portable C	61'783	113'745	114'270	228'015
This work	12'428 (2.39 ms)	23'338 (4.49 ms)	23'400 (4.50 ms)	46'738 (8.99 ms)
Speedup	5.0 x	4.9 x	4.9 x	4.9 x
SIKEp751				
Portable C	110'838	179'540	193'048	372'589
This work	21'908 (4.21 ms)	37'700 (7.25 ms)	37'560 (7.22 ms)	75'260 (14.47 ms)
Speedup	5.1 x	4.8 x	5.1 x	5.0 x

Compared with baseline, the assembly versions are 4.3, 4.2, 4.9 and 5.0 times faster for `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`, respectively. The different speedups have two reasons: First, larger parameter sets benefit more from vectorization than the smaller ones. This is because the proportional overhead from limbification and delimbification decreases the more operations are performed in limbs. Second, the better the limbs are utilized, the better the improvement one can expect. In our case we accommodate integers up to a length of $4 \cdot \lceil \log_2(p) \rceil$ to allow lazy reductions. Split into 56-bit limbs, one needs 8, 10, 11 and 14 limbs for `SIKEp434`, `SIKEp503`, `SIKEp610` and `SIKEp751`, respectively. The limb utilization of `SIKEp434` and `SIKEp610` is more beneficial than the one of `SIKEp503` and `SIKEp751`, leading to a higher speedup.

$\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ performance We further break down the SIKE performance to the arithmetic over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$. The results are available in Table 2.

We observe a significant speedup in each of the optimized operations. Similarly as in the overall results, the improvements are most accentuated with larger parameter sets. Arithmetic operating in 56-limbs (multiplication and squaring in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$) further benefits most if the limbs are well utilized.

Table 2: Performance (in cycles) of $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic functions on an IBM z15 LPAR at 5.2 GHz. In brackets are speedup factors compared to the portable C baseline implementation.

	SIKEp434	SIKEp503	SIKEp610	SIKEp751
$\mathbb{F}_p$ `add`	31 (3.0 x)	31 (3.7 x)	31 (4.7 x)	36 (5.6 x)
$\mathbb{F}_p$ `sub`	31 (2.2 x)	31 (2.5 x)	31 (3.2 x)	31 (3.8 x)
$\mathbb{F}_p$ `mul`	161 (3.4 x)	254 (3.0 x)	312 (3.5 x)	405 (3.8 x)
$\mathbb{F}_p$ `rdc`	88 (3.0 x)	109 (3.4 x)	140 (3.6 x)	171 (4.2 x)
$\mathbb{F}_{p^2}$ `add`	46 (3.4 x)	46 (4.7 x)	62 (4.8 x)	72 (5.4 x)
$\mathbb{F}_{p^2}$ `sub`	46 (2.7 x)	46 (3.8 x)	62 (3.5 x)	72 (3.6 x)
$\mathbb{F}_{p^2}$ `mul`	390 (4.4 x)	514 (4.3 x)	603 (5.3 x)	842 (5.3 x)
$\mathbb{F}_{p^2}$ `sqr`	286 (4.1 x)	410 (3.9 x)	499 (4.7 x)	702 (4.7 x)

Table 3: Performance (in thousands of cycles) of encaps+decaps with assembly-optimized implementations on different platforms: IBM z15, Intel x64 (Skylake), ARMv8-A, ARMv7-M.

	IBM z15	Intel x64 ([3])	ARMv8-A ([3])	ARMv7-M ([9])
`SIKEp434`	17'818	20'024	58'542	184'000
`SIKEp503`	28'667	27'959	81'621	257'000
`SIKEp610`	46'738	54'699	181'107	493'000
`SIKEp751`	75'260	84'553	282'089	770'000

Comparison with other platforms SIKE is an attractive target for assembly-optimized implementations on several architectures. The SIKE round 3 submission bundles and reports some of the fastest software implementations available in [3]. While it is difficult to compare cycle counts across CPU architectures, we summarize the values reported in related literature in Table 3 to give an overview. The assembly-optimized version targeting Intel x64 uses ADX and MULX instructions available from the Broadwell architecture onwards. Compared with our z15 implementation, we achieve an improved cycle count of up to a factor 1.17 (for `SIKEp610`). Our speedup factor in CPU cycles is mainly influenced by the utilization of the 56-bit limbs, which is best in `SIKEp610` and worst in `SIKEp503`. The assembly-optimized version from [3] and [10] targets 64-bit ARMv8 Cortex-A platforms. Compared to the implementation for this target, the cycle count is up to 3.88 times improved on z15. Another optimized version from [9] targets the embedded ARMv7 Cortex-M4 architecture. The cycle count of our z15 implementation is up to 10.3 times improved. We further note that z15 is operating at 5.2 GHz and therefore often has a further advantage in absolute timings. Another interesting architecture for vectorized optimization is AVX512-IFMA, providing 52-bit fused multiply and add (FMA) instructions on 512-bit wide registers, where [8] predicts a potential speedup of a factor 1.72 compared to a non-vectorized implementation.

4 Optimizing Dilithium

Dilithium is a digital signature scheme based on the hardness of the MLWE and MSIS lattice problems. The arithmetic used in Dilithium operates over the polynomial ring $\mathbb{Z}_q[X]/(X^{256}+1)$. To perform efficient operations over this ring the number-theoretic transform (NTT) is used. The NTT consists of modular operations modulo the prime $q = 2^{23} - 2^{13} + 1$. The other operations that consume a great deal of computational time are the generation of longer sequences of random values from seeds.

4.1 Modular Multiplication

All of the multiplications done as part of the NTT, inverse-NTT and pointwise operations perform a centered reduction modulo q. The centered reduction of element r reduces the element to the range $-\frac{q-1}{2} \leq r' \leq \frac{q-1}{2}$. The reference implementation provides the function `montgomery_reduce(int64_t a)` to perform this reduction. The IBM Z vector architecture provides 32-bit multiplication instructions which can either produce a full 64-bit product or produce the high or low half of the product. The most efficient implementation for doing the modular multiplication can be shown in Algorithm 4.

Algorithm 4: Multiplication with centered reduction

```
1: function MontMul($a = (a_0, a_1, a_2, a_3)$, $b = (b_0, b_1, b_2, b_3)$)
2:     $plo$   VMLF$(a, b)$
3:     $phi$   VMHF$(a, b)$
4:     $t$   VMLF$(plo, qinv)$
5:     $t$   VMHF$(t, q)$
6:     $c$   $phi - t$
7:     return $c$
8: end function
```

By using Algorithm 4, the timing of a single 256-element pointwise multiplication was reduced from 525 ns to 56 ns, more than a 9x speedup.

4.2 NTT

To get the greatest performance from the NTT, as many elements as possible should be contained within vector registers. The IBM Z vector architecture has 32 128-bit registers which can hold a combined 128 32-bit elements. However, some of these registers are necessary to hold temporary values so fewer are available for use. With manual register allocation in the butterflies of the NTT, we found that we could use 16 of the vector registers holding 64 elements at a time. By loading groups of elements into each register that are spaced by 16 elements, we can perform 4 levels of the NTT without having to reload any registers. We iterate four times to complete all of the first 4 levels of the Cooley-Tukey butterflies while saving out the results when finished. To complete the final 4 levels, we load 64 consecutive elements at a time and can to complete all remaining levels before storing the final results. We iterate four times to complete the NTT.

When we combined Algorithm 4 to perform the multiplications by the twiddle factors and the optimized traversal of the elements, the performance improved from 3475 ns for an NTT with the reference implementation to 236 ns for the NTT in this work, more than a 14x speedup.

Similar optimizations were performed for the Gentleman-Sande butterflies of the inverse-NTT improving performance from 8270 ns to 255 ns, a speedup of over 32x.

4.3 Hashing

Dilithium makes extensive use of extendible output functions (XOFs) to expand seeds. There are two variants of Dilithium, one uses the SHAKE128 and SHAKE256 XOFs, and the other uses AES256-CTR to construct the XOF. IBM Z is the only commercially available CPU with support for full Keccak acceleration. ARMv8.4 added several instructions to help improve the performance of doing a round of Keccak, but the operations are still broken down into a sequence of instructions.

On IBM Z when the message-security-assist 6 is installed, the Compute Intermediate Message Digest (KIMD) and Compute Last Message Digest (KLMD) instructions perform an entire SHAKE operation with a single invocation. The KIMD instruction is used when you need to make multiple calls with various input buffers. The KLMD instruction can be used either to complete an operation started with KIMD by applying padding if necessary and then storing the output, or can be used when the entire input buffer is available. Since we always have the entire input buffer at the time of the operation, our implementation only uses the KLMD instruction. Because this instruction performs the entire SHAKE operation it has some startup overhead. To avoid this overhead when possible, the existing interface of initializing the random stream and then squeezing blocks out of the stream was changed: the interface to do both the stream initialization and squeezing the blocks of output as a single operation. Also, in order to avoid having to call the squeeze block interface if after rejection sampling there was not enough data, we increased the size of the initial amount of data obtained in order to reduce the frequency of having to request more random bits.

This same interface was also reused when implementing the variant of Dilithium that uses AES256-CTR. Instead of using the KLMD instruction, the Cipher Message with Authentication (KMA) instruction was used. This instruction is usually used for accelerating AES-GCM operations, but it is also the fastest way to perform AES-CTR mode operations. Since the KMA instruction is designed to encrypt data, a zero buffer that is the size of the output buffer must be created and passed to the instruction. The GCM tag is discarded at the end of the operation.

4.4 Sampling

Once a random bit stream has been obtained from either SHAKE or AES256-CTR, the output is then sampled. The sampling that is done the most is rejection sampling on the XOF output in the ExpandA function. This sampling takes three bytes of input data, masks off the upper 9 bits, and then compares the resulting value to q. If the value is less than q, it is written to the output. If it is not, another three bytes of input are obtained. While it may seem that this is a serial operation, we vectorized the operation in a similar to the AVX2 implementation, sampling 4 input values at a time.

This vectorized sampling, along with the hashing enhancements above for NIST level-2 using SHAKE 128 the ExpandA function, the execution time was reduced from $48'600$ µs to $7'350$ µs, over a 6.5x speedup. When using NIST level-2 and AES256-CTR, execution time was reduced from $135'200$ µs to $4'300$ µs. The AES256-CTR is still faster than the SHAKE128 implementation. Some of the slow down is due to the KLMD interface having to save and restore the Keccak sponge state which is almost half the size of the XOF output, reducing the possible throughput. The KMA instruction only has to load the 32-byte key and initial counter value and saves the final counter value.

4.5 Implementation and Results

We implemented optimized versions of the Dilithium parameter sets for NIST security levels 2, 3, and 5 for both the SHAKE and AES variants. The library is based on the official Dilithium reference third round submission [6]. For evaluation we measured a baseline and an optimized version:

1. A baseline portable C implementation based on the reference code provided to NIST.
2. An optimized version with vectorized polynomial operations, including hand coded assembly versions of the NTT and inverse-NTT, vectorized sampling for the ExpandA function as well as accelerated SHAKE and AES256-CTR functions.

The code was compiled on Linux on Z (Fedora 31) using GCC 9.2.1 with `-march=arch13 -O3 -mzvector` compiler flags added to the reference set of flags. The evaluation machine was an IBM z15 logical partition (LPAR) operating at 5.2 GHz. The results are available in Table 4.

Table 4: Performance (in cycles) of Dilithium on an IBM z15 LPAR at 5.2 GHz.

	`KeyGen`	`Sign`	`Verify`
Dilithium2			
Portable C (ref)	684’841	3’102’625	763’919
This work	104’000 (20.0 µs)	253’239 (48.7 µs)	93’080 (17.9 µs)
Speedup	6.6 x	12.3 x	8.2 x
Dilithium2-AES			
Portable C (ref)	1’241’346	3’939’394	1’231’936
This work	84’760 (16.3 µs)	222’565 (42.8 µs)	76’440 (14.7 µs)
Speedup	14.6 x	17.7 x	16.1 x
Dilithium3			
Portable C (ref)	1’213’252	5’231’388	1’217’799
This work	239’201 (46.0 µs)	419’118 (80.6 µs)	142’999 (27.5 µs)
Speedup	5.1 x	12.5 x	8.5 x
Dilithium3-AES			
Portable C (ref)	2’362’562	6’878’307	2’053’712
This work	201’238 (38.7 µs)	367’647 (70.7 µs)	112’321 (21.6 µs)
Speedup	11.7 x	18.7 x	18.3 x
Dilithium5			
Portable C (ref)	1’748’487	5’842’697	1’861’797
This work	266’762 (51.3 µs)	538’191 (103.5 µs)	234’519 (45.1 µs)
Speedup	6.6 x	10.9 x	7.9 x
Dilithium5-AES			
Portable C (ref)	3’608’605	8’163’265	3’466’667
This work	204’362 (39.3 µs)	458’109 (88.1 µs)	177’317 (34.1 µs)
Speedup	17.7 x	17.8 x	19.6 x

5 Conclusion and further work

We have demonstrated a significant speedup of SIKE and Dilithium by using vector and SHA3 instructions on IBM z15. Our results are competitive with other platforms and in many cases outperform them in terms of cycle count and absolute timings. SIKE benefits especially from VMSL instructions on z15 that allow an efficient implementation of finite field and extension field arithmetic. Dilithium also benefits from vector instructions to speedup modular multiplication, NTT and sampling. The application of hardware SHA3 acceleration further greatly improves the performance of default SHAKE-based Dilithium versions.

As future work, we envision to apply some of the optimization methods to other platforms with advanced instruction sets like AVX512. Our optimizations in SIKE will further also have an application to emerging isogeny-based schemes, such as the signature scheme SQISign [5]. Our optimized Dilithium implementation will presumably benefit in future hardware generations from even faster SHA3 hardware. Other lattice-based schemes and the hash-based signature SPHINCS+-SHAKE256 that are mainly dominated by Keccak related functions can be speed up with these instructions as well.

References

[1] z/Architecture Principles of Operation. https://www.ibm.com/support/pages/zarchitecture-principles-operation, 2019. Accessed: 2021-04-19.

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the second round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2020.

[3] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, David Jao, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation, 2020.

[4] Luca De Feo, David Jao, and Jérôme Plût. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. *Journal of Mathematical Cryptology*, 8(3):209–247, 2014.

[5] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 64–93. Springer, Heidelberg, December 2020.

[6] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[7] Armanndo Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie–Hellman key exchange protocol. *IEEE Transactions on Computers*, 67:1622–1636, November 2017.

[8] D. Kostic and S. Gueron. Using the new vpmadd instructions for the new post quantum key encapsulation mechanism sike. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 215–218, 2019.

[9] H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh. Supersingular isogeny key encapsulation (sike)round 2 on arm cortex-m4. *IEEE Transactions on Computers*, pages 1–1, 2020.

[10] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. Optimized sike round 2 on 64-bit arm. In Ilsun You, editor, *Information Security Applications*, pages 341–353, Cham, 2020. Springer International Publishing.

[11] Christopher Tozzi. 9 mainframe statistics that may surprise you. https://www.precisely.com/blog/mainframe/9-mainframe-statistics, 2020. Accessed: 2021-04-19.

[12] James You, Qi Zhang, Curtis D'Alves, Bill O'Farrell, and Christopher Kumar Anand. Using z14 fused-multiply-add instructions to accelerate elliptic curve cryptography. In *CASCON*, pages 284–291, 2019.