SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange

Hwajeong Seo[1], Zhe Liu[2], Patrick Longa[3] and Zhi Hu[4]

[1] Hansung University, South Korea
hwajeong84@gmail.com
[2] Nanjing University of Aeronautics and Astronautics, China
sduliuzhe@gmail.com
[3] Microsoft Research, USA
plonga@microsoft.com
[4] School of Mathematics and Statistics, Central South University, China
huzhi_math@csu.edu.cn

Abstract. We present high-speed implementations of the post-quantum supersingular isogeny Diffie-Hellman key exchange (SIDH) and the supersingular isogeny key encapsulation (SIKE) protocols for 32-bit ARMv7-A processors with NEON support. The high performance of our implementations is mainly due to carefully optimized multiprecision and modular arithmetic that finely integrates both ARM and NEON instructions in order to reduce the number of pipeline stalls and memory accesses, and a new Montgomery reduction technique that combines the use of the `UMAAL` instruction with a variant of the hybrid-scanning approach. In addition, we present efficient implementations of SIDH and SIKE for 64-bit ARMv8-A processors, based on a high-speed Montgomery multiplication that leverages the power of 64-bit instructions. Our experimental results consolidate the practicality of supersingular isogeny-based protocols for many real-world applications. For example, a full key-exchange execution of SIDHp503 is performed in about 176 million cycles on an ARM Cortex-A15 from the ARMv7-A family (i.e., 88 milliseconds @2.0GHz). On an ARM Cortex-A72 from the ARMv8-A family, the same operation can be carried out in about 90 million cycles (i.e., 45 milliseconds @1.992GHz). All our software is protected against timing and cache attacks. The techniques for modular multiplication presented in this work have broad applications to other cryptographic schemes.

Keywords: Post-quantum cryptography, SIDH, SIKE, Montgomery multiplication, ARM, NEON.

1 Introduction

Modular multiplication is one of the performance-critical building blocks of many public-key cryptographic schemes. Although efficient techniques such as Montgomery multiplication [28] have been widely studied for many years in an effort to engineer faster cryptographic implementations on different CPU architectures [22, 6, 16, 31, 8, 24, 29, 30], speeding up the modular multiplication operation still remains an important challenge, especially for computing-intensive cryptographic schemes.

One particularly attractive example of those computing-intensive schemes is the increasingly popular supersingular isogeny Diffie-Hellman key exchange (SIDH) protocol proposed by Jao and De Feo in 2011 [19]. SIDH is the basis of the supersingular isogeny key encapsulation (SIKE) protocol [4], which is currently under consideration by the National Institute of Standards and Technology (NIST) for inclusion in a future standard for post-quantum cryptography [32]. One of the attractive features of SIDH and SIKE is their relatively small public keys which are, to date, the most compact among well-established quantum-resistant algorithms. On the downside, these protocols are much slower than other popular candidates for post-quantum cryptography, such as those based on ideal lattices. Therefore, speeding up modular multiplication, which is a fundamental arithmetic operation in SIDH, has become one of the critical tasks from which depends the practicality of these isogeny-based cryptographic schemes.

In this work, we focus on optimizing the modular multiplication, with special focus on SIDH and SIKE, for the popular 32-bit ARMv7-A and 64-bit ARMv8-A architectures, which are widely used in smartphones, mini-computers, wearable devices, and many others. Many advanced ARMv7-A processors include the NEON engine, which comprises Single Instruction Multiple Data (SIMD) instructions capable of processing heavy multimedia workloads with high efficiency. In order to exploit the parallel computing power of SIMD instructions, traditional serial implementations need to be rewritten in a vectorized way. One approach is to use so-called *redundant* representations to guarantee that sums of partial results fit in "vector" registers, as suggested in [18]. However, this can be inconvenient in some settings in which vectorized implementations of low-level arithmetic operations need to be integrated into libraries that use canonical (i.e., non-redundant) representations for the upper layers. In this work, one important goal is to facilitate the integration of modular arithmetic functions into the SIDH library [11]. This state-of-the-art supersingular isogeny-based library implements the SIDH and SIKE protocols with support for two parameter sets based on the primes $\text{p503} = 2^{250}3^{159} - 1$ and $\text{p751} = 2^{372}3^{239} - 1$. Accordingly, we adopt *non-redundant* representations for all the proposed algorithms and implementations targeting these parameters sets.

Some works in the literature have studied algorithms for modular arithmetic based on non-redundant representations. The basic idea of these methods is to split modular arithmetic operations in two or more parts, such that intermediate computations do not overflow. For example, in [8] Bos et al. introduced a novel implementation of Montgomery multiplication using vector instructions that flipped the sign of the precomputed Montgomery constant and accumulated the result in two separate intermediate values that are computed in parallel. In [26, 27], Martins and Sousa proposed a product-scanning based Montgomery multiplication that computes a pair of 32-bit multiplications at once. One critical disadvantage of all these algorithms is the high number of Read-After-Write (RAW) dependencies, which makes the execution flow suboptimal. In order to minimize these RAW dependencies, Seo et al. [29] introduced a novel 2-way Cascade Operand Scanning (COS) multiplication that performs the partial products of two modular multiplications in an interleaved way. The method was further improved in [30] by using the additive Karatsuba method for integers as long as 1024 and 2048 bits.

In this work, we take a different approach for the case of 32-bit ARMv7-A processors. We split operations into smaller sub-products either using Karatsuba (for multiprecision multiplication) or schoolbook multiplication (for Montgomery reduction), and then distribute these to the ARM and NEON engines for processing. Since the goal is to minimize pipeline stalls due to data hazards, ARM and NEON instructions are carefully interleaved to maximize the instruction level parallelism (ILP). In each case, initial and final computations are processed using ARM instructions. This approach favors the use of a non-redundant representation, which makes representation conversion routines unnecessary and facilitates integration into existing libraries, as pursued. We then specialize these techniques to Montgomery multiplication over the "SIDH-friendly" primes p503 and p751 to realize high-speed implementations of the SIDH and SIKE protocols.

Finally, we also report efficient implementations of SIDH and SIKE for 64-bit ARMv8-A processors, based on a high-speed Montgomery multiplication that leverages the power of 64-bit instructions.

Our main contributions can be summarized as follows:

1. We propose a unified ARM/NEON multiprecision multiplication for 32-bit ARMv7-A processors that finely integrates ARM and NEON instructions to exploit ARM's instruction level parallelism. We show that this approach reduces the number of memory accesses by employing both ARM and NEON registers for temporary storage, and reduces the number of pipeline stalls even in processors with out-of-order execution capabilities, such as the ARM Cortex-A15 CPU.

2. We introduce a new Montgomery reduction algorithm for 32-bit ARMv7-A processors that combines the use of the `UMAAL` instruction with a variant of the hybrid-scanning approach. We then use this algorithm to engineer a *generic* Montgomery reduction that combines the use of ARM and NEON instructions.

3. We specialize the new Montgomery reduction for ARMv7-A to the setting of supersingular isogeny-based protocols using the primes p503 and p751.

4. We design efficient multiprecision multiplication and Montgomery reduction algorithms specialized to the setting of supersingular isogenies using the prime p503 for 64-bit ARMv8-A.

The proposed *non-redundant* modular arithmetic algorithms, which were implemented on top of the most recent version of Microsoft's SIDH library [11] (version 3.0), demonstrates that the supersingular isogeny-based protocols SIDH and SIKE are practical for the myriad of applications that use 32-bit and 64-bit ARM-powered devices. For example, a full key-exchange using SIDHp503 is performed in about 176 million cycles on an ARM Cortex-A15 from the ARMv7-A family (i.e., 88 milliseconds @2.0GHz). The same computation is executed in about 90 million cycles on an ARM Cortex-A72 from the ARMv8-A family (i.e., 45 milliseconds @1.992GHz). Our software does not use conditional branches over secret data or secret memory addresses and, hence, is protected against timing and cache attacks.

Our software for 64-bit ARMv8-A processors has been integrated to the SIDH library which is available in `https://github.com/Microsoft/PQCrypto-SIDH`.

The remainder of this paper is organized as follows. In Section 2, we briefly describe the 32-bit ARMv7-A and 64-bit ARMv8-A architectures. Sections 3 and 4 describe the proposed multiprecision multiplication and Montgomery reduction algorithms, respectively, and Section 5 presents the specialized Montgomery multiplications for the setting of supersingular isogeny-based protocols. Thereafter, we summarize our experimental results on several ARMv7-A and ARMv8-A processors in Section 6, and conclude the paper in Section 7.

2 ARM architecture

With over 100 billion ARM-based chips shipped worldwide as of 2017 [1], ARM has consolidated its hegemony as the most popular instruction set architecture (ISA) in terms of quantity. In this work, we focus on the popular 32-bit ARMv7-A and 64-bit ARMv8-A architecture families, which belong to the "application" profile implemented by cores from the Cortex-A series.

2.1 ARMv7-A architecture

As other traditional 32-bit ARM architectures, the ARMv7-A ISA [2] is equipped with 16 32-bit registers (`R0`∼`R15`), from which 14 are available for use, and an instruction set supporting 32-bit operations or, in the case of Thumb and Thumb2, a mix of 16- and 32-bit operations. Since the maximum capacity of the register set is of only 448 bits, it is of critical importance to engineer techniques and algorithms that use efficiently the available registers in order to minimize memory accesses.

ARMv7-A processors support powerful unsigned integer multiplication instructions. Our implementation of modular multiplication makes use of the following multiply instructions:

- `UMULL` (unsigned multiplication):
  `UMULL R0, R1, R2, R3` computes $(\texttt{R1} \parallel \texttt{R0}) \leftarrow \texttt{R2} \times \texttt{R3}$.
- `UMLAL` (unsigned multiplication with accumulation):
  `UMLAL R0, R1, R2, R3` computes $(\texttt{R1} \parallel \texttt{R0}) \leftarrow (\texttt{R1} \parallel \texttt{R0}) + \texttt{R2} \times \texttt{R3}$.
- `UMAAL` (unsigned multiplication with double accumulation):
  `UMAAL R0, R1, R2, R3` computes $(\texttt{R1} \parallel \texttt{R0}) \leftarrow \texttt{R1 + R0 + R2} \times \texttt{R3}$.

Of particular interest is the `UMAAL` instruction, which performs a 32×32-bit multiplication followed by accumulations with two 32-bit values (note that the result can be held in a 64-bit register without producing an overflow). This instruction achieves the same latency and throughput of the other two multiply instructions, which means that accumulation is virtually executed for free.

A multiply instruction that is followed by a multiply-and-add with a dependency on the accumulator triggers a special accumulator forwarding that produces that both instructions are issued back-to-back. For example, on Cortex-A8 and Cortex-A9 cores the multiply and multiply-and-add instructions listed above have a latency of 5 cycles. If the special accumulator forwarding is exploited, a series of multiply and multiply-and-add instructions can achieve a throughput of 3 cycles per instruction.

NEON engine for ARMv7-A. Many ARMv7-A cores include NEON [2], a powerful 128-bit SIMD engine that comes with 16 128-bit quadruple-word registers (`Q0`∼`Q15`) which can also be viewed as 32 64-bit double-word registers (`D0`∼`D31`). NEON includes support for 8-, 16-, 32- and 64-bit signed and unsigned integer operations that are executed in a vectorized fashion.

Our implementation of modular multiplication makes extensive use of the following two multiply instructions:

- `VMULL.U32` (vectorized unsigned multiplication):
  `VMULL.U32 Q0, D2, D3[0]` computes $\texttt{D1} \leftarrow \texttt{D2[1]} \times \texttt{D3[0]}, \texttt{D0} \leftarrow \texttt{D2[0]} \times \texttt{D3[0]}$.
- `VMLAL.U32` (vectorized unsigned multiplication with accumulation):
  `VMLAL.U32 Q0, D2, D3[0]` computes $\texttt{D1} \leftarrow \texttt{D1 + D2[1]} \times \texttt{D3[0]}, \texttt{D0} \leftarrow \texttt{D0 + D2[0]} \times \texttt{D3[0]}$.

The instructions `VMULL.U32` and `VMLAL.U32` have a latency of 6 cycles. However, if any of these two instructions is followed by a multiply-and-add instruction (in this case, by `VMLAL.U32`) that depends on the result of the first instruction then the instructions are issued back-to-back thanks to a special accumulator forwarding. In this case, a series of multiply and multiply-and-add instructions can achieve a throughput of 2 cycles per instruction.

2.2 ARMv8-A architecture

ARMv8-A, or simply ARMv8, is the latest generation of ARM architectures targeted at the "application" profile. It includes the typical 32-bit architecture, called "AArch32", and a 64-bit architecture named "AArch64" with its associated instruction set "A64" [3]. AArch32 preserves backwards compatibility with ARMv7 and supports the so-called "A32" and "T2" instructions sets, which correspond to the traditional 32-bit and Thumb instruction sets, respectively. AArch64 comes equipped with 31 general purpose 64-bit registers (`X0`∼`X31`), and an instruction set supporting 32-bit and 64-bit operations. The significant register expansion means that with AArch64 the maximum register capacity is expanded to 1,984 bits, a 4x increase with respect to ARMv7.

ARMv8-A processors started to dominate the smartphone market soon after their first release in 2011, and nowadays they are widely used in various smartphones (e.g., iPhone and Samsung Galaxy series). Since this architecture is used primarily in embedded systems and smartphones, efficient and compact implementations are of special interest.

ARMv8-A supports powerful unsigned integer multiplication instructions. Our implementation of modular multiplication uses the AArch64 architecture and makes extensive use of the following multiply instructions:

- `MUL` (unsigned multiplication, low part):
  `UMULL X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`) mod 2^{64}.
- `UMULH` (unsigned multiplication, high part):
  `UMULH X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`)/2^{64}.

The two instructions above are required to compute a full 64-bit multiplication of the form 128-bit $\leftarrow 64 \times 64$-bit, namely, the `MUL` instruction computes the lower 64-bit half of the product while `UMULH` computes the higher 64-bit half.

3 Multiprecision multiplication

There is a plethora of works in the literature that study the use of NEON instructions to implement multiprecision multiplication or the full Montgomery multiplication on 32-bit ARMv7-A processors [8, 26, 27, 29, 30]. However, we point out that it is possible to achieve relatively good performance by exploiting efficient ARM instructions, such as `UMAAL`, especially if one limits the analysis to the use of non-redundant representations.

In Table 1, we summarize the results of our experiments with state-of-the-art techniques to realize ARM-only and NEON-only implementations of 256- and 512-bit multiprecision multiplications. For the ARM-only implementation we adapted the implementation by Fujii and Aranha [14] that uses the `UMAAL` instruction with the Consecutive Operand Caching (COC) method. In the case of NEON-only multiplication, we adapted the implementation by Seo et al. [30] that uses Karatsuba multiplication with the sub-products carried out with the Cascade Operand Scanning (COS) method proposed in [29].

Table 1: Comparison of 256-bit and 512-bit multiprecision multiplications using non-redundant representations and implemented with either ARM or NEON instructions (32-bit ARMv7-A architecture). The results (in clock cycles) were obtained on an ARM Cortex-A15 processor.

Bitlength	Reference	Instruction	Timings [cc]
256-bit	[14]	ARM	158
	[30]	NEON	188
512-bit	[14]	ARM	596
	[30]	NEON	632

The results in Table 1 show that, in the case of methods using non-redundant representations, ARM-based implementations of multiprecision multiplication can be more efficient than their NEON-based counterparts. Next, we propose an approach for multiprecision multiplication that takes into account this observation. Moreover, in contrast to most previous works, the proposed method finely mixes ARM and NEON instructions to improve performance further.

3.1 Unified ARM/NEON multiplication for ARMv7-A

A well-known technique to improve CPU usage in certain ARM processors consists of exploiting the instruction-level parallelism of ARM and NEON instructions. For example, in cryptography the idea of mixing ARM and NEON instructions has been exploited to speed up the Salsa20 stream cipher [5]. In [12], Faz et al. used the technique to optimize the arithmetic over the extension field $\mathbb{F}_{p^2}$ by interleaving ARM-only and NEON-only multiplication and modular reduction routines. And Longa [25] engineered implementations of extension field multiplication and squaring using NEON that were interleaved with other less expensive operations such as additions and subtractions over $\mathbb{F}_p$ implemented with ARM instructions.

In the following, we will use m to denote the maximum bitlength of the input operands to the multiprecision multiplication (we refer to the corresponding operation as an m-bit multiplication). In order to simplify the description we will assume that m is some even integer value.

In this section, we show how to use the ARM/NEON mixing technique with the Karatsuba multiplication for realizing the multiprecision multiplication. For this, some Karatsuba sub-products are implemented with ARM instructions and others with NEON instructions, and these different routines are then finely interleaved. Sub-products run by ARM are implemented with the `UMULL/UMAAL` instructions and the Consecutive Operand Caching (COC) method [14], while sub-products run by the NEON engine are implemented with the `VMULL/VMLAL` instructions and the Cascade Operand Scanning (COS) method [29]. After corresponding computations are concluded, the results are combined together using ARM instructions. The proposed unified ARM/NEON multiplication is depicted in Algorithm 1. Additional details follow below.

Algorithm 1 Unified ARM/NEON multiplication

```
Input: Two $m$-bit operands $A = (A_H || A_L)$ and $B = (B_H || B_L)$
Output: $2m$-bit result $C$
1: $A_M \leftarrow |A_L - A_H|$                                   {ARM}
2: $B_M \leftarrow |B_L - B_H|$                                   {ARM}

   Interleaved section begin
3: $C_L \leftarrow A_L \cdot B_L$                                  {ARM}
4: $C_M \leftarrow A_M \cdot B_M$                                  {NEON}
5: $C_H \leftarrow A_H \cdot B_H$                                  {ARM}
   Interleaved section end

6: return $C \leftarrow C_L + (C_L + C_H - C_M) \cdot 2^{\frac{m}{2}} + C_H \cdot 2^m$     {ARM}
```

The method applies 1-level Karatsuba at the top layer [20]. Accordingly, we split an m-bit operand multiplication into $m/2$-bit operand multiplications, which reduces the cost of multiplication from four $m/2$-bit multiplications to three $m/2$-bit multiplications. There are two well-known approaches to realize Karatsuba, namely *additive* and *subtractive*. Assume a multiplication of two operands $A = A_H \cdot 2^{\frac{m}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{m}{2}} + B_L$. The multiplication $C = A \cdot B$ can be computed according to the following equation with the additive Karatsuba algorithm:

$$A_H \cdot B_H \cdot 2^m + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{m}{2}} + A_L \cdot B_L$$

And with the subtractive Karatsuba algorithm [17]:

$$A_H \cdot B_H \cdot 2^m + [A_H \cdot B_H + A_L \cdot B_L - |A_H - A_L| \cdot |B_H - B_L|] \cdot 2^{\frac{m}{2}} + A_L \cdot B_L$$

In our implementations, we use the subtractive Karatsuba algorithm, which avoids the carry bits in the computation of C_M but requires two absolute differences and one conditional negation (see Algorithm 1). As shown in Algorithm 1, the computations of the two absolute differences are performed by the ARM processor at the beginning of the execution, and then the results are directly transferred to NEON registers. Thus, the corresponding $m/2$-bit multiplication is assigned to the NEON engine, and the remaining two $m/2$-bit multiplications are assigned to the ARM processor. These NEON and ARM routines are finely interleaved to exploit the instruction-level parallelism. As stated before, we use the COC method with `UMAAL` for the ARM routines, while the COS method is used for the NEON routine. A word-level (32-bit) depiction of the algorithm for a 256-bit multiprecision multiplication is presented in Figure 1.

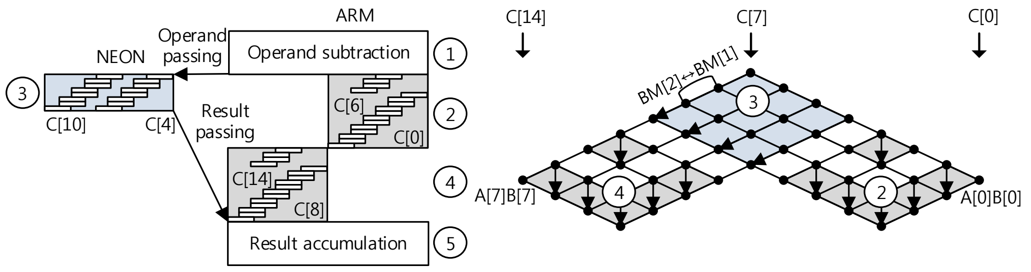

Figure 1: Unified 256-bit ARM/NEON multiplication at the word-level, ①: operand subtraction; ② ④: two sub-products on ARM; ③: sub-product on NEON; ⑤: result accumulation.

The method is depicted using the execution structure from Algorithm 1 and also in rhombus form (see Figure 1), where the following notation is required. Let A and B be operands of length m bits each. Each operand is written as $A = (A[n-1], ..., A[1], A[0])$ and $B = (B[n-1], ..., B[1], B[0])$, where $n = \lceil m/w \rceil$ is the number of words to represent operands, and w is the computer word size. The result $C = A \cdot B$ is represented by $C = (C[2n-1], ..., C[1], C[0])$. In the rhombus, the lowest indices (i, $j = 0$) of the product appear at the rightmost corner, whereas the highest indices (i, $j = n-1$) appear at the leftmost corner. A black arrow over a point indicates the processing of a partial product. The lowermost points represent the results $C[i]$ from the rightmost corner ($i = 0$) to the leftmost corner ($i = 2n-1$). For the ARM routines —executed as Single Instruction Single Data (SISD)— one partial product is performed per row, whereas for the NEON routine —executed as Single Instruction Multiple Data (SIMD)— two partial products are performed per row.

The sub-sections corresponding to the three Karatsuba sub-products are denoted by ②, ④ (ARM using COC method), and ③ (NEON using COS method). The execution in Figure 1 is as follows.

- In Step ①, the operand subtraction to generate the middle results $A_M[0 \sim 3] \leftarrow |A[0 \sim 3] - A[4 \sim 7]|$ and $B_M[0 \sim 3] \leftarrow |B[0 \sim 3] - B[4 \sim 7]|$ is performed using ARM instructions.

- After Step ①, the interleaved section begins. In Step ③, the middle part of Karatsuba multiplication ($C_M[0 \sim 7] \leftarrow A_M[0 \sim 3] \times B_M[0 \sim 3]$) is performed on the NEON engine, while in Steps ② and ④ the lower and higher parts (resp.) of Karatsuba are performed by ARM instructions.

- Straightforward Karatsuba multiplication requires $3n$ addition/subtraction operations for the calculation of $(C_L + C_H - C_M) \cdot 2^{\frac{m}{2}}$; see Algorithm 1. We store partially accumulated intermediate results in order to save some computations and memory accesses. First, the higher and lower parts are accumulated and stored as $T \leftarrow C_L + C_H \cdot 2^{\frac{m}{2}}$, which allows us to save $n/2$-word additions, n-word load and n-word save operations. Second, the intermediate results of ARM are accumulated together with the intermediate results of NEON as follows: $C \leftarrow T_L + (((T_L + T_M) + (T_M + T_H) \cdot 2^{\frac{m}{2}}) + C_{ML}) \cdot 2^{\frac{m}{2}}$, where $T_L \leftarrow T \bmod 2^{\frac{m}{2}}$, $T_M \leftarrow (T \bmod 2^m) \text{ div } 2^{\frac{m}{2}}$, $T_H \leftarrow T \text{ div } 2^m$, and $C_{ML} \leftarrow C_M \bmod 2^{\frac{m}{2}}$. The intermediate result C_{ML} is directly transferred from NEON to ARM registers, saving n-word memory accesses ($n/2$-word load and $n/2$-word save operations).
- Lastly, the remaining part from the NEON computation (i.e., $C_{MH} \leftarrow C_M \text{ div } 2^{\frac{m}{2}}$) is used to compute $C \leftarrow C + (C_{MH} + T_H) \cdot 2^{\frac{3m}{2}}$ in Step ⑤. The intermediate result C_{MH} is also directly transferred from NEON to ARM registers, which saves n-word memory accesses ($n/2$-word load and $n/2$-word save operations).

Even though ARM processors such as the Cortex-A15 support out-of-order execution, we observed experimentally that manual scheduling significantly improves performance. Since the number of lines of the ARM routines is roughly twice the number of lines of the NEON routine, we mixed two ARM instructions per each NEON instruction (i.e., `...; ARM instruction; ARM instruction; NEON instruction; ...`, and so on). For example, for the 512-bit multiplication the straight version without interleaving is computed in 590 cycles on a Cortex-A15. After interleaving as explained above, the cost is reduced to only 470 cycles (i.e., multiplication is executed 1.25x faster).

In Table 2, we compare the results of the proposed method for 512-and 768-bit multiprecision multiplication with other implementations based on ARM and NEON instructions. For example, the proposed unified ARM/NEON approach is about 1.3x faster than methods based on ARM-only or NEON-only instructions for computing 512-bit multiprecision multiplication.

Table 2: Comparison of implementations of 512-bit and 768-bit multiprecision multiplication on an ARM Cortex-A15 processor (32-bit ARMv7-A architecture). Timings are reported in clock cycles.

Bit Length	Method	Instruction	Timings [cc]
512-bit	Fujii et al. [14]	ARM	596
	GMP-6.1.2 [15]	ARM	1,138
	Seo et al. [30]	NEON	632
	This work	ARM/NEON	470
768-bit	GMP-6.1.2 [15]	ARM	2,408
	This work	ARM/NEON	912

3.2 Multiprecision multiplication for ARMv8-A

The extended register space and the new support for operations with 64-bit arguments introduce a significant performance improvement on 64-bit ARMv8-A processors. However, it can also be observed a widening in the gap between the cost of multiply instructions and the cost of other simple instructions such as addition. We summarize the performance of a few representative instructions on the ARM Cortex-A72 processor in Table 3. While it is possible to execute instructions such as addition and subtraction at an optimal rate of 2 instructions per cycle, a full 64×64-bit multiplication operation can only run at a rate of $4 + 3 = 7$ cycles per operation (recall that to obtain a 128-bit product, one requires a `MUL` instruction to compute the lower 64-bit half of the product while `UMULH` computes the higher 64-bit half; cf. Section 2.2).

Table 3: Performance comparison of various instructions on the ARM Cortex-A72 processor (64-bit ARMv8-A architecture). Timings are in clock cycles.

AArch64 instruction	Instruction group	Latency [cc]	Throughput [cc]
`ADD/ADC/SUB/SBC`	ALU, basic	1	2
`MUL`	multiply	3	1/3
`UMULH`	multiply high	6	1/4

Based on the observation above we engineer a multiprecision multiplication that minimizes the use of multiplication instructions by making extensive use of Karatsuba multiplication. Specifically, we use two-level additive Karatsuba to implement a 512-bit multiprecision multiplication. At the lowest level, we implement a 128 × 128-bit multiplication using the Comba method [9] based on the following multiplication/addition instruction sequence

```
 ⋮
MUL    X0,X4,X5
UMULH  X1,X4,X5
ADDS   X10,X10,X2
ADCS   X11,X11,X3
ADC    X12,XZR,XZR

MUL    X2,X6,X7
UMULH  X3,X6,X7
ADDS   X10,X10,X0
ADCS   X11,X11,X1
ADC    X12,X12,XZR
 ⋮
```

where the last accumulation sequence performs the computation $(\texttt{X12}\|\texttt{X11}\|\texttt{X10}) \leftarrow (\texttt{X1}\|\texttt{X0}) + (\texttt{X12}\|\texttt{X11}\|\texttt{X10})$. Note that the two registers `X0` and `X1` hold the lower and higher parts of the product `X4` × `X5`, respectively, which are computed by the first multiplication pair placed in the upper sequence. This is done in order to hide the cost of the addition instructions using the multiply instructions. Thus, each 5-instruction multiplication/addition sequence requires 7 clock cycles (of reciprocal throughput) to execute on an ARM Cortex-A53 or Cortex-A72 processor. This means that a 128-bit Comba multiplication can be executed in approximately 28 cycles. Experimental results of this approach in the context of SIDH and SIKE are reported in Section 6.

4 Modular reduction

Multiprecision modular multiplication is a performance-critical building block in pre-quantum (e.g. RSA) and post-quantum (e.g. SIDH) cryptography. One of the most well-known techniques used for its implementation is Montgomery reduction [28]. A basic description is depicted in Algorithm 2.

The efficient implementation of Montgomery multiplication has been actively studied for both SISD and SIMD architectures. For the case of SISD architectures, such as 8-bit AVR and some 32-bit ARM processors, one of the most efficient approaches is the hybrid Montgomery multiplication. This method organizes the computation in a

Algorithm 2 Montgomery reduction

```
Require: An odd modulus $M$, the Montgomery radix $R > M$, an operand $T \in [0, M^2 - 1]$,
    and the pre-computed constant $M' = -M^{-1} \bmod R$
Ensure: Montgomery product $Z = \text{MonRed}(T, R) = T \cdot R^{-1} \bmod M$
1: $Q \leftarrow T \cdot M' \bmod R$
2: $Z \leftarrow (T + Q \cdot M)/R$
3: if $Z \geq M$ then $Z \leftarrow Z - M$
4: return Z
```

two-level routine comprising an inner and an outer loop [16, 24]. One can choose different approaches to implement each routine, but recent hybrid implementations have utilized the product-scanning method for both routines. When it comes to SIMD architectures, such as ARM with NEON, some works [30, 29, 23] have employed the COS method, issuing two multiplications at once and finely re-ordering the computation routines to avoid pipeline stalls.

One crucial difference with previous works is that we exploit both ARM and NEON instructions to implement the Montgomery reduction. For the NEON part we adopt the COS method. However, for the ARM part, neither hybrid-scanning nor COC appears to be efficient. On one hand the use of the `UMAAL` instruction favors a row-wise (operand scanning) execution. On the other hand, the COC method does not facilitate an efficient interleaved implementation with NEON. To overcome these issues we propose a more efficient variant of the hybrid-scanning approach using the `UMAAL` instruction. The details are depicted in Algorithm 3.

Unlike previous hybrid approaches, the outer loop of the proposed algorithm applies operand-scanning (for loop in line 2) while the inner loop applies product-scanning (for loop in line 15). For simplicity and efficiency, the algorithm is described for the case in which the operand-scanning width is 3 and the word length of the operand is divisible by 3 [1]; but these settings can be easily modified.

We now show how to exploit the new hybrid-scanning method to realize a more efficient Montgomery reduction that mixes ARM and NEON instructions. In the following, we fix the bitlength of the Montgomery radix to $s = wn$, where w is the computer word size and $n = \lceil m/w \rceil$ is the number of words needed to represent an m-bit modulus M.

As stated before, the NEON part can be implemented efficiently with the COS method. For the ARM part, we use the proposed variant of the hybrid-scanning method. The proposed unified ARM/NEON algorithm for Montgomery reduction is detailed in Algorithm 4. As can be seen, after the initial multiplication performed by the ARM processor, the results are transferred to NEON registers. Then, the $s/2$-bit multiplications with the lower half of the modulus M (i.e., with M_L) are performed with the new hybrid-scanning method using the ARM instruction set, while the remaining two $s/2$-bit multiplications (with M_H) are performed with the COS method using NEON. These NEON and ARM routines are finely interleaved to exploit the instruction-level parallelism.

A word-level depiction of the algorithm with an 8 32-bit word input operand is presented in Figure 2 using the execution structure from Algorithm 4 and also in rhombus form, where the following notation is required. Let T be the input operand, written as $T = (T[2n-1], ..., T[1], T[0])$, Q be the result in line 1 of Algorithm 4, written as $Q = (Q[n-1], ..., Q[1], Q[0])$, and M be the modulus, written as $M = (M[n-1], ..., M[1], M[0])$. As before, the lowest indices (i, $j = 0$) of the product appear at the rightmost corner of the rhombus, whereas the highest indices (i, $j = n-1$) appear at the leftmost corner. A black arrow over a point indicates the processing of a partial product. The lowermost

[1] The operand-scanning width set to 3 ensures the efficient use of `UMAAL` with a limited number of 32-bit general purpose registers for the column-wise multiplication without the need of carry handling.

Algorithm 3 Hybrid-scanning algorithm for Montgomery reduction targeting 32-bit ARMv7-A processors with support for the `UMAAL` instruction. Operand-scanning width is set to 3

```
Input: n-word modulus $M = (m_{n-1}, ..., m_1, m_0)$ with $3|n$, operand $T = (t_{2n-1}, ..., t_1, t_0)$
    with $T < M^2 - 1$, and pre-computed constant $m'_0 = -m_0^{-1} \bmod 2^w$ with $w = 32$
Output: Montgomery residue $Z = T \cdot 2^{-wn} \bmod M$
1: $(p, u, v) \leftarrow (0, 0, 0)$
2: for $i$ from 0 by 3 to $n - 1$ do
3:     $q_i \leftarrow t_i \cdot m'_0 \bmod 2^w$                                  {MUL}
4:     $(p, u, v) \leftarrow m_0 \cdot q_i + t_i + (p, u, v)$                       {UMAAL}
5:     $(p, u, v) \leftarrow (0, p, u)$
6:     $(p, u, v) \leftarrow m_0 \cdot q_{i+1} + t_{i+1} + (p, u, v)$               {UMAAL}
7:     $q_{i+1} \leftarrow v \cdot m'_0 \bmod 2^w$                               {MUL}
8:     $(p, u, v) \leftarrow m_1 \cdot q_i + (p, u, v)$                           {UMAAL}
9:     $(p, u, v) \leftarrow (0, p, u)$
10:    $(p, u, v) \leftarrow m_0 \cdot q_{i+2} + t_{i+2} + (p, u, v)$               {UMAAL}
11:    $(p, u, v) \leftarrow m_1 \cdot q_{i+1} + (p, u, v)$                        {UMAAL}
12:    $q_{i+2} \leftarrow v \cdot m'_0 \bmod 2^w$                               {MUL}
13:    $(p, u, v) \leftarrow m_2 \cdot q_i + (p, u, v)$                           {UMAAL}
14:    $(p, u, v) \leftarrow (0, p, u)$
15:    for $j$ from 3 by 1 to $n - 1$ do
16:        $(p, u, v) \leftarrow m_{j-1} \cdot q_{i+2} + t_{i+j} + (p, u, v)$       {UMAAL}
17:        $(p, u, v) \leftarrow m_j \cdot q_{i+1} + (p, u, v)$                    {UMAAL}
18:        $(p, u, v) \leftarrow m_{j+1} \cdot q_{i+0} + (p, u, v)$                {UMAAL}
19:        $t_{i+j} \leftarrow v$
20:        $(p, u, v) \leftarrow (0, p, u)$
21:    $(p, u, v) \leftarrow m_{n-2} \cdot q_{i+2} + t_{i+n} + (p, u, v) + (c_1, c_0)$   {UMAAL}
22:    $(p, u, v) \leftarrow m_{n-1} \cdot q_{i+1} + (p, u, v)$                     {UMAAL}
23:    $t_{i+n} \leftarrow v$
24:    $(p, u, v) \leftarrow (0, p, u)$
25:    $(p, u, v) \leftarrow m_{n-1} \cdot q_{i+2} + t_{i+n+1} + (p, u, v)$          {UMAAL}
26:    $t_{i+n+1} \leftarrow v$
27:    $(p, u, v) \leftarrow (0, p, u)$
28:    $(c_1, c_0) \leftarrow (u, v)$
29: for $j$ from 0 by 1 to $n - 2$ do
30:    $z_j \leftarrow p_{j+n}$
31: $(z_n, z_{n-1}) \leftarrow (c_1, c_0)$
32: if $Z \geq M$ then
33:    $Z \leftarrow Z - M$
34: return $Z$
```

Algorithm 4 Unified ARM/NEON Montgomery reduction

```
Input: An odd $m$-bit modulus $M = (M_H \| M_L)$, the Montgomery radix $R = 2^s$, where
    $s = wn$ with $w = 32$ and $n = \lceil m/w \rceil$, an operand $T \in [0, M^2 - 1]$, and pre-computed
    constant $M' = -M^{-1} \bmod R$
Output: $m$-bit Montgomery product $Z = T \cdot R^{-1} \bmod M$
1: $Q = (Q_H \| Q_L) \leftarrow T \cdot M' \bmod R$                          {ARM}

   Interleaved section begin
2: $Z_1 \leftarrow M_L \cdot Q_L$                                             {ARM}
3: $Z_2 \leftarrow M_H \cdot Q_L$                                             {NEON}
4: $Z_3 \leftarrow M_L \cdot Q_H$                                             {ARM}
5: $Z_4 \leftarrow M_H \cdot Q_H$                                             {NEON}
   Interleaved section end

6: $Z \leftarrow (T + Z_1 + (Z_2 + Z_3) \cdot 2^{\frac{s}{2}} + Z_4 \cdot 2^s)/2^s$          {ARM}

7: if $Z \geq M$ then
8:     $Z \leftarrow Z - M$                                                   {ARM}
9: return $Z$
```

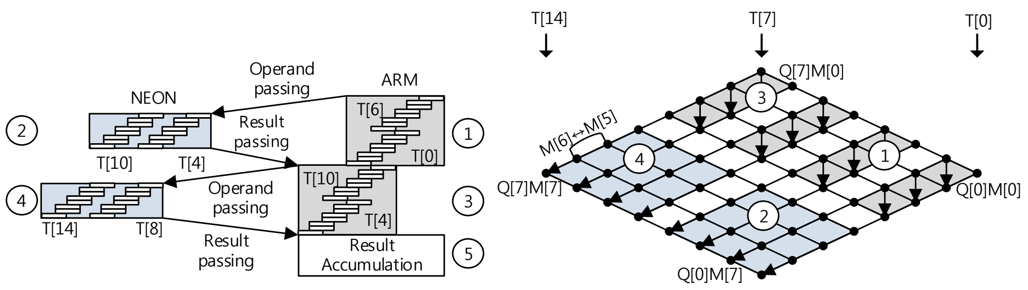

Figure 2: Unified 256-bit ARM/NEON Montgomery reduction at the word-level (8-word operand), ① ③: two sub-products on ARM; ② ④: two sub-products on NEON; ⑤: result accumulation.

points represent the results $Z[i]$ from the rightmost corner ($i = 0$) to the leftmost corner ($i = 2n - 1$). For the ARM routines —executed as Single Instruction Single Data (SISD)— one partial product is performed per row, whereas for the NEON routine —executed as Single Instruction Multiple Data (SIMD)— two partial products are performed per row. The sub-sections in the “interleaved” section are denoted by ①③, computed by the ARM processor using hybrid-scanning, and ②④, computed by the NEON engine using COS.

5 Modular multiplication for SIDH

In this section, we adapt the techniques described in previous sections to implement modular multiplication for the supersingular isogeny-based protocols SIDH and SIKE. Specifically, we target the parameter sets based on the primes p503 and p751 [10, 4]. For more information about SIDH, SIKE and their efficient implementation, the reader is referred to [19, 10, 4, 11].

Multiprecision modular multiplication is a basic and central operation for the implementation of SIDH [19, 10]. In this setting, Montgomery multiplication can be efficiently exploited and further simplified by taking advantage of so-called Montgomery-friendly modulus. There are three cases that admit efficient computation:

- When the lower word of the modulus is $2^w - 1$ (i.e., `0xFFFFFFFF` for $w = 32$), the pre-computed Montgomery constant $m' = -m_0^{-1} \bmod 2^w$ is equal to 1. This optimizes away the multiplication $T \cdot m'$ in a radix-2^w Montgomery multiplication.
- Multiplications with an *all-ones* word can be replaced by a simpler operation with shifts and subtractions (e.g., $T \times$ `0xFFFFFFFF` $\rightarrow T \times 2^{32} - T$).
- Assuming a modulus $M + 1$ turns the lower part of the modulus into *all-zero* words, which directly eliminates several multiplications and additions/subtractions.

The optimizations above, which were first pointed out by Costello et al. [10] in the setting of SIDH when using moduli of the form $2^x \cdot 3^y - 1$ (referred to as "SIDH-friendly" primes), are exploited by the SIDH library to reduce the complexity of Montgomery reduction from $\mathcal{O}(n^2 + n)$ to roughly $\mathcal{O}(\frac{n}{2}^2)$ [11].

The advantage of using Montgomery multiplication for SIDH-friendly primes was also recently confirmed by Bos and Friedberger [7], who studied and compared different approaches, including Barrett reduction. In [7] they also explored fast modular arithmetic for generalized moduli of the form $2^x p^y \pm 1$ with $p \geq 3$. Their most efficient results are reported for the case $p = 19$. However, it is still unclear how this parameter choice affects the performance of the full SIDH protocol. Karmakar et al. [21] proposed a modular reduction algorithm based on the radix $2^{\frac{x}{2}} \cdot 3^{\frac{y}{2}}$. Unfortunately, their technique has an interleaved execution and only works efficiently with moduli of the form $2 \cdot 2^x \cdot 3^y$ with x and y even (e.g., this modulus shape is incompatible with the primes proposed for SIDH and SIKE [10, 4]). Bos and Friedberger fixed these issues with a variant of the $2^{\frac{x}{2}} 3^{\frac{y}{2}}$-radix approach, but their method requires on-the-fly conversion of the inputs to these special radix representations.

Based on this analysis, we choose Montgomery multiplication to implement SIDH-friendly modular arithmetic, following [10, 7, 13]. Moreover, we further reduce the complexity of modular multiplication by employing the unified ARM/NEON approach.

As previously stated, we focus on efficient algorithms that have inputs/outputs expressed in non-redundant representation. This is done with the goal of facilitating integration of our field arithmetic implementation into existing libraries, such as the SIDH library. Between interleaved and non-interleaved Montgomery multiplication, the latter is the preferred approach for SIDH since this favors the use of Karatsuba and lazy reduction at the $\mathbb{F}_{p^2}$ level. Accordingly, for the multiplication part we use the multiprecision multiplication algorithm described in Section 3.1, in which two $s/2$-bit multiplications are executed by ARM instructions using the COC method, and one $s/2$-bit multiplication is executed by NEON using the COS method. For the Montgomery reduction part, we adapt the Montgomery reduction algorithm described in Section 4 to SIDH-friendly primes. In this case, one $s/2$-bit multiplication is executed with ARM instructions using our variant of the hybrid-scanning method, and one $s/2$-bit multiplication is executed by NEON using the COS method. The method is detailed in Algorithm 5. As in [10], our approach uses the transformed modulus $\tilde{M} = M + 1$ during intermediate computations in order to convert the lower words of p503 and p751 to *all-zero* words and, thus, save a significant number of multiplications and additions.

A word-level depiction of the algorithm with an 8 32-bit word input operand is presented in Figure 3 using the execution structure from Algorithm 5 and also in rhombus form. The execution flow is divided into three sub-sections (①②③). The sub-sections in the "interleaved" section, namely ① and ②, are computed by the ARM processor using hybrid-scanning and by the NEON engine using COS, respectively. Note that we assume that the lower words of the transformed modulus $\tilde{M} = (M[n-1], ..., M[1], M[0])$ are all zeroes; i.e., $\tilde{M}[0] \sim \tilde{M}[3] \leftarrow 0$ in Figure 3, which eliminates half of the multiplications in the rhombus.

Algorithm 5 Unified ARM/NEON Montgomery reduction for SIDH-friendly primes

```
Input: $\tilde{M} = M + 1 = (\tilde{M}_H \| \tilde{M}_L)$ for an odd $m$-bit modulus $M$, the Montgomery radix $R = 2^s$, where $s = wn$ with $w = 32$ and $n = \lceil m/w \rceil$, an operand $T \in [0, M^2 - 1]$, and pre-computed constant $M' = -M^{-1} \bmod R$
Output: $m$-bit Montgomery product $Z = T \cdot R^{-1} \bmod M$
1: Set $Q = (Q_H \| Q_L) \leftarrow T \cdot M' \bmod 2^s$

   Interleaved section begin
2: $T \leftarrow T + (\tilde{M}_H \cdot Q_L) \cdot 2^{\frac{s}{2}}$                {ARM}
3: $Z_4 \leftarrow \tilde{M}_H \cdot Q_H$                                  {NEON}
   Interleaved section end

4: $Z \leftarrow (T + Z_4 \cdot 2^s - Q)/2^s$                          {ARM}
5: if $Z \geq M$ then
6:     $Z \leftarrow Z - M$                                           {ARM}
7: return $Z$
```

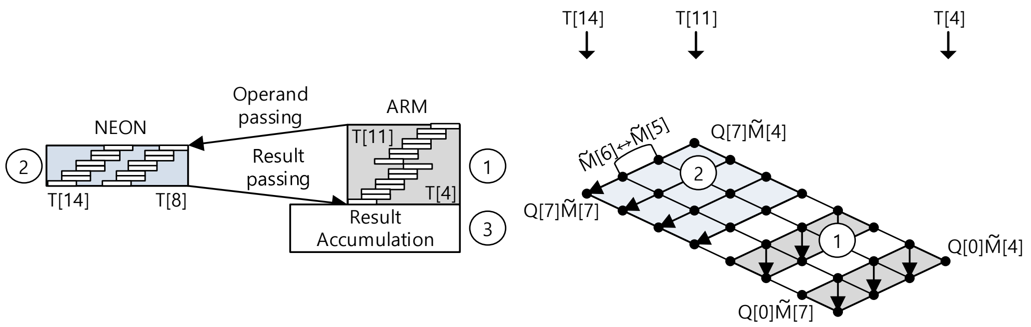

Figure 3: Unified 256-bit ARM/NEON Montgomery reduction at the word-level (8-word operand) for SIDH-friendly primes, ①: sub-product on ARM; ②: sub-product on NEON; ③: result accumulation.

6 Performance evaluation

In this section, we evaluate the performance of the proposed algorithms for 32-bit ARMv7-A and 64-bit ARMv8-A processors. All our implementations were written in assembly language and compiled with GCC 5.4.0 with optimization level `-O3`.

First, we implemented generic 512-bit and 768-bit Montgomery multiplications using the multiprecision multiplication algorithm described in Section 3.1 and the modular reduction algorithm described in Section 4. Table 4 summarizes our results an compares them with other efficient implementations on an ARM Cortex-A15 from the ARMv7-A family. The column "Approach" indicates whether the corresponding Montgomery multiplication implementation interleaves the multiplication and reduction routines. As can be seen, in the 512-bit case Fujii et al. [14] achieves the highest efficiency among ARM-only implementations, while Seo et al.'s COS method achieves the best performance in the case of NEON [30]. Our implementation exploiting mixed ARM/NEON instructions is roughly 1.3x and 1.4x faster than [14] and [30], respectively.

Table 4: Comparison of different 512-bit and 768-bit Montgomery multiplication implementations. Results (in clock cycles) were obtained on an ARM Cortex-A15.

Bit Length	Instruction	Method	Approach	Timings [cc]
512-bit	ARM	GMP-6.1.2 [15]	-	2,900
		Bos et al. [8]	Interleaved	2,373
		Fujii et al. [14]	Separated	1,332
	NEON	Martins et al. [26, 27]	Interleaved	4,206
		Bos et al. [8]	Interleaved	2,473
		Seo et al. [29]	Interleaved	1,485
		Seo et al. [30]	Separated	1,408
	ARM/NEON	This work	Separated	1,034
768-bit	ARM	GMP-6.1.2 [15]	Separated	5,561
	ARM/NEON	This work	Separated	2,006

SIDH on ARMv7-A. To evaluate the performance of the proposed Montgomery multiplication specialized to the setting of SIDH (cf. Section 5), we integrate it into the SIDH library [11], version 3.0. This library implements the SIDH and SIKE protocols using the parameters sets SIDHp503 and SIDHp751 (resp. SIKEp503 and SIKEp751) based on the 503-bit and 751-bit primes p503 and p751, respectively [10, 4].

Table 5 shows the results of software implementations of the SIDHp503 and SIDHp751 protocols on ARM Cortex-A15. We include implementation results from Koziel et al. [23] which presented an implementation of Montgomery multiplication using NEON and the COS method from [30]. In comparison, our implementation of Montgomery multiplication for p503 is about 1.8x faster than Koziel et al.'s. This speedup directly reflects on the full protocol: an SIDHp503 key exchange is executed approximately 1.7x faster. In the case of p751, Koziel et al.'s only reported an implementation using generic C. In this case, SIDHp751 runs approximately 2.8x faster with our assembly-optimized ARM/NEON based Montgomery multiplication.

As reference, we also include results of the *unoptimized* reference implementation written in C using the Microsoft SIDH library [11]. We mention than in this case the proposed implementation is roughly 13x faster for the computation of a full SIDH key exchange.

SIDH on ARMv8-A. For this case, we implemented the multiprecision multiplication algorithm described in Section 3.2, and the *shifted* 2^{Bw}-radix Montgomery reduction

Table 5: Comparison of implementations of the SIDHp503 and SIDHp751 protocols on ARM Cortex-A15 (ARMv7-A architecture). Timings are reported in terms of clock cycles.

Implementation	Language	Instruction	Timings [cc]	Timings [$cc \times 10^6$]				
			$\mathbb{F}_p$ mul	Alice R1	Bob R1	Alice R2	Bob R2	Total
SIDHp503								
SIDH v3.0 [11]	C	generic	8,947	597	657	487	555	2,296
Koziel et al. [23]	ASM	NEON	1,372	83	87	66	68	302
This work	ASM	ARM/NEON	780	46	50	38	42	176
SIDHp751								
SIDH v3.0 [11]	C	generic	36,592	2,006	2,256	1,650	1,924	7,836
Koziel et al. [23]	C	generic	N/A	437	474	346	375	1,632
This work	ASM	ARM/NEON	1,502	150	170	120	144	584

proposed by [7], with $B = 4$. As in the previous section, we integrated our implementation of the Montgomery multiplication for ARMv8-A into the SIDH library [11], version 3.0.

Table 6 summarizes the results of different software implementations of the SIDHp503 protocol on two ARMv8-A processors: (i) a 1.512GHz ARM Cortex-A53 processor with in-order execution, and (ii) a 1.992GHz ARM Cortex-A72 processor with out-of-order capability. We include results for the ARMv8-A implementation by Campagna [4], which was submitted to the NIST post-quantum standardization process [32] as an "additional implementation" in the SIKE submission package [4]. As can be seen, our implementation of Montgomery multiplication for the 503-bit prime is up to 1.2x faster than the assembly-optimized implementation by Campagna. Moreover, a full SIDHp503 key exchange based on our efficient multiplication is 1.23x and 1.27x faster on the Cortex-A72 and Cortex-A53 processors, respectively. Our results are similar for the SIKE protocol (see Table 7): our implementations are between 1.23x and 1.28x faster on the same ARMv8-A processors for computing SIKEp503's encapsulation and decapsulation operations.

As reference, we also include results of the *unoptimized* reference implementation written in C using the Microsoft SIDH library [11]. In this case, the proposed implementation is between 5 and 6 times faster for the computation of the SIDH and SIKE protocols.

Table 6: Comparison of implementations of the SIDHp503 protocol on ARMv8-A based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Processor	Timings [cc]	Timings [$cc \times 10^6$]				
			$\mathbb{F}_p$ mul	Alice R1	Bob R1	Alice R2	Bob R2	Total
SIDH v3.0 [11]	C	Cortex-A53	4,453	167.2	136.2	184.5	155.9	643.8
Campagna [4]	ASM		1,187	44.0	35.9	48.7	41.2	169.8
This work	ASM		971	34.5	28.1	38.3	32.4	133.3
SIDH v3.0 [11]	C	Cortex-A72	3,942	149.1	121.5	164.3	139.4	574.3
Campagna [4]	ASM		865	28.8	23.4	31.7	26.9	110.8
This work	ASM		753	23.4	19.1	25.9	21.9	90.3

Table 7: Comparison of implementations of the SIKEp503 protocol on ARMv8-A based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Processor	Timings [cc]	Timings [$cc \times 10^6$]			
			$\mathbb{F}_p$ mul	KeyGen	Encaps	Decaps	Total
SIDH v3.0 [11]	C	Cortex-A53	4,453	184.5	303.3	323.0	626.3
Campagna [4]	ASM		1,187	48.8	80.0	85.3	165.3
This work	ASM		971	38.4	62.7	66.9	129.6
SIDH v3.0 [11]	C	Cortex-A72	3,942	164.4	270.6	287.9	558.5
Campagna [4]	ASM		865	31.8	52.2	55.6	107.8
This work	ASM		753	25.9	42.5	45.3	87.8

Finally, we remark that the comparisons above do not take into account energy and power consumption. The analysis of these metrics, especially for the proposed ARM/NEON algorithms in comparison with ARM-only and NEON-only implementations, is particularly relevant but requires further study. For example, while NEON instructions consume more power than regular instructions their use may reduce code density and execution time, which in turn can have a positive effect on power reduction. Moreover, mixing ARM and NEON instructions, as done in our algorithms, reduces the number of power-expensive memory accesses. This analysis is left as future work.

7 Conclusion

This paper presented unified ARM/NEON algorithms for high-speed Montgomery multiplication on 32-bit ARMv7-A processors, and an efficient implementation of Montgomery multiplication for 64-bit ARMv8-A processors. A combination of these optimizations yields very efficient Montgomery multiplications that are shown, for example, to be approximately 1.3x faster than the previously best implementations on an ARM Cortex-A15 processor. We integrated our fast modular arithmetic implementations into Microsoft's SIDH library and reported the fastest performance on 32-bit and 64-bit ARM processors to date. For example, a full key-exchange execution of SIDHp503 is performed in about 88 milliseconds on a 2.0GHz ARM Cortex-A15 from the ARMv7-A family. On a 1.992GHz ARM Cortex-A72 from the ARMv8-A family, the same operation can be carried out in about 45 milliseconds. These results, which push further the performance of post-quantum supersingular isogeny-based protocols, are 1.7x and 1.2x faster than the previously fastest assembly-optimized implementations of SIDH/SIKE on the same Cortex-A15 and Cortex-A72 processors, respectively.

Acknowledgments. We would like to thank Erkay Savas and the reviewers for their valuable and helpful feedback. Hwajeong Seo was partially supported by the National Research Foundation of Korea (NRF) with grant funded by the Korea Government (MSIT) (No. NRF-2017R1C1B5075742) and by the Institute for Information & Communications Technology Promotion (IITP) with grant funded by the Korea Government (MSIT) (No. 2018-0-00264, Research on Blockchain Security Technology for IoT Services). Zhe Liu (corresponding author of this paper) was supported by the Natural Science Foundation of Jiangsu Province of China (No. SBK2018043466) and Fundamental Research Funds for the Central Universities (Grant No. NE2018106). Zhi Hu was partially supported by the Natural Science Foundation of China (Grant No. 61602526).

References

[1] ARM Holdings. Q1 2017 roadshow slides. https://www.arm.com/company/-/media/arm-com/company/Investors/Quarterly%20Results%20-%20PDFs/Arm_SB_Q1_2017_Roadshow_Slides_Final.pdf, 2017.

[2] ARM Limited. ARM architecture reference manual ARMv7-A and ARMv7-R edition. https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf, 2007–2014.

[3] ARM Limited. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf, 2013–2017.

[4] R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes,

V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, 2017. Available at https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip.

[5] D. J. Bernstein and P. Schwabe. NEON crypto. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.

[6] T. Blum and C. Paar. Montgomery modular exponentiation on reconfigurable hardware. In *IEEE Symposium on Computer Arithmetic (ARITH '99)*. IEEE, 1999.

[7] J. W. Bos and S. Friedberger. Fast arithmetic modulo $2^x p^y \pm 1$. In *IEEE Symposium on Computer Arithmetic (ARITH'17)*, pages 148–155. IEEE, 2017.

[8] J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. In *Selected Areas in Cryptography - SAC 2013*, pages 471–489. Springer, 2013.

[9] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[10] C. Costello, P. Longa, and M. Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In M. Robshaw and J. Katz, editors, *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.

[11] C. Costello, P. Longa, and M. Naehrig. SIDH Library. https://github.com/Microsoft/PQCrypto-SIDH, 2016–2018.

[12] A. Faz-Hernández, P. Longa, and A. H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015.

[13] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Transactions on Computers (to appear)*, 2017.

[14] H. Fujii and D. F. Aranha. Curve25519 for the Cortex-M4 and beyond. *Progress in Cryptology - LATINCRYPT 2017*, 2017.

[15] GMP. The GNU Multiple Precision Arithmetic Library. Available for download at https://gmplib.org/, 2018.

[16] J. Großschädl, R. M. Avanzi, E. Savas, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005.

[17] M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. *J. Cryptographic Engineering*, 5(3):201–214, 2015.

[18] Intel. Using streaming SIMD extensions (SSE2) to perform big multiplications. https://software.intel.com/sites/default/files/14/4f/24960, 2006.

[19] D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In B. Yang, editor, *Post-Quantum Cryptography (PQCrypto 2011)*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.

[20] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 1962.

[21] A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient finite field multiplication for isogeny based post quantum cryptography. In *International Workshop on the Arithmetic of Finite Fields*, pages 193–207. Springer, 2016.

[22] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[23] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani. NEON-SIDH: efficient implementation of supersingular isogeny Diffie-Hellman key exchange protocol on ARM. In *International Conference on Cryptology and Network Security (CANS 2016)*, pages 88–103. Springer, 2016.

[24] Z. Liu and J. Großschädl. New speed records for Montgomery modular multiplication on 8-bit AVR microcontrollers. In *International Conference on Cryptology in Africa (Africacrypt 2014)*, pages 215–234. Springer, 2014.

[25] P. Longa. Four$\mathbb{Q}$NEON: faster elliptic curve scalar multiplications on ARM processors. In R. Avanzi and H. M. Heys, editors, *Selected Areas in Cryptography - SAC 2016*, volume 10532 of *Lecture Notes in Computer Science*, pages 501–519. Springer, 2017.

[26] P. Martins and L. Sousa. On the evaluation of multi-core systems with SIMD engines for public-key cryptography. In *Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pages 48–53. IEEE, 2014.

[27] P. Martins and L. Sousa. Stretching the limits of programmable embedded devices for public-key cryptography. In *Workshop on Cryptography and Security in Computing Systems*, page 19. ACM, 2015.

[28] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[29] H. Seo, Z. Liu, J. Großschädl, J. Choi, and H. Kim. Montgomery modular multiplication on ARM-NEON revisited. In *Information Security and Cryptology (ISC 2014)*, pages 328–342. Springer, 2014.

[30] H. Seo, Z. Liu, J. Großschädl, and H. Kim. Efficient arithmetic on ARM-NEON and its application for high-speed RSA implementation. *Security and Communication Networks*, 9(18):5401–5411, 2016.

[31] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer, 2008.

[32] The National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization, 2017–2018. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.