Faster Modular Arithmetic For Isogeny Based Crypto on Embedded Devices

Joppe W. Bos[1] and Simon J. Friedberger[1,2]

[1] NXP Semiconductors, Leuven, Belgium, joppe.bos@nxp.com
[2] KU Leuven - iMinds - COSIC, Leuven, Belgium, simon.friedberger@esat.kuleuven.be

Abstract. We show how to implement the Montgomery reduction algorithm for isogeny based cryptography such that it can utilize the *unsigned multiply accumulate accumulate long* instruction present on modern ARM architectures. This results in a practical speed-up of a factor 1.34 compared to the approach used by SIKE: the supersingular isogeny based submission to the ongoing post-quantum standardization effort.
Moreover, motivated by the recent work of Costello and Hisil (ASIACRYPT 2017), which shows that there is only a moderate degradation in performance when evaluating large odd degree isogenies, we search for more general supersingular isogeny friendly moduli. Using graphics processing units to accelerate this search we find many such moduli which allow for faster implementations on embedded devices. By combining these two approaches we manage to make the modular reduction 1.5 times as fast on a 32-bit ARM platform.

Introduction

Recent advances in quantum computing [10, 21, 27] resulted in a call for proposals for new public-key cryptography standards by the National Institute of Standards and Technology (NIST) [26, 6]. The selection procedure for this new cryptographic standard has started and has further accelerated the research of *post-quantum* cryptography schemes. One of the candidates for new schemes is based on the hardness of constructing a smooth-degree isogeny, between two supersingular elliptic curves defined over a finite field [16]. The original proposal is called Supersingular Isogeny Diffie-Hellman (SIDH). It forms the basis of the Supersingular Isogeny Key Encapsulation protocol (SIKE) [1] submitted to NIST. SIKE additionally includes optimizations from recent works such as [8, 13]. The exact details of this protocol are outside the scope of this paper.

More research related to the security and practical performance of these schemes is needed to make an informed decision in the selection process for future standards. This paper is related to the latter: we are interested in enhancing the practical performance of SIKE on embedded processors. This is especially important given the current trend, in the era of Internet of Things (IoT), to interconnect more and more small devices.

Motivation. There are two main arithmetic operations in SIDH-based protocols. First, one needs to compute elliptic curve scalar multiplications where the curve is defined over a quadratic extension of a finite field $\mathbf{F}_q$ with $q = 2^x p^y - 1$ and prime. Hence, all the curve arithmetic boils down to arithmetic in $\mathbf{F}_q$. Second, evaluations of ℓ-isogenies for $\ell \in \{2, p\}$ are required. In the proposed SIKE protocol, and all previous implementations, $p = 3$ is used.

The community has looked into different techniques to optimize the arithmetic modulo q [8, 20, 5]. In [5] the option to chose $p > 3$ is investigated and moduli are given which result in modular reduction an implementation which is about 12 % faster than the one used in [8]. However, this has an impact on the p-isogeny evaluations and it was unclear how large one could choose p. Recently, Costello and Hisil studied this in more detail [7] and introduced new formulas for computing arbitrary odd-degree isogenies between elliptic curves in Montgomery form. Their results show that there is only a moderate degradation in performance when evaluating $(2d + 1)$-isogenies as d grows. Hence, larger values of d can now be considered practical and ready to be used in SIDH implementations.

This work is motivated by this recent result and the following practical setting. We assume the use of an SIDH prime modulus $\hat{q} = 2^x m - 1$, where m is the product of small primes. In the SIKE protocol, one party computes a 2^x-isogeny as a composition of 2- and 4-isogenies [16, 8] while the other party computes an m-isogeny as a composition of p_i-isogenies where p_i are the small primes dividing m. If the modular arithmetic modulo $\hat{q}$ is faster than arithmetic modulo $q = 2^x p^y - 1$ then the party computing the 2^x isogeny, which can be an embedded IoT device, gets the best of both worlds: fast modular arithmetic and fast isogeny evaluations. The server can make use of the faster modular arithmetic but has to evaluate p_i-isogenies which is slightly more expensive than evaluating p-isogenies for small p. However, servers can often afford a small performance price especially when this allows public-key cryptography on massively deployed embedded devices.

Contributions. The main contributions of this paper are twofold. First, we investigate how the modular arithmetic used in supersingular isogeny instantiated protocols can be accelerated. The implementations of SIKE [1] implement the Montgomery reduction algorithm in product scanning form.[3] This approach allows for an efficient implementation and manages to keep most values in registers. However, it seems non-trivial to make use of the unsigned multiply accumulate accumulate long instruction, present on ARMv6 and above, which multiplies two 32-bit values and adds two 32-bit values to the result in a single instruction. In Section 3.1 we show how to compute the Montgomery reduction differently which does allow using this arithmetic instruction and accelerates the modular reduction in practice.

Second, we search for more generic SIDH-friendly moduli of the form $2^x m - 1$ which have additional properties which allow for more efficient implementations on 32-bit embedded devices. This approach is described in Section 4. Using the computational power of graphics processing units we manage to find over half a million SIDH moduli, for a range of security levels, where at least one of the 32-bit digits of m is zero. Since a multiplication by m is the main arithmetic cost of a Montgomery reduction this accelerates embedded implementations. The performance impact of both optimizations is presented in Section 5.

1 Preliminaries

Throughout this paper we typically assume that a b-bit non-negative multi-precision integer X is represented by an array of $n = \lceil b/r \rceil$ computer words as $X = \sum_{i=0}^{n-1} x_i r^i$ (the so-called *radix-r representation*), where $r = 2^w$ for the word size w of the target computer architecture and $0 \leq x_i < r$. Unless stated otherwise we target the 32-bit ARM architecture (i.e. $w = 32$). Here x_i is the i-th computer word (or digit) of the integer X.

1.1 Montgomery reduction

Montgomery presented a way to compute modular reduction *without* computing any expensive division operations [25]. The main idea is to adjust the representation of the integers used and

[3] In previous work this approach is often credited to an unpublished 1995 work by Scott [30]. However, this work does not seem to be available online anymore. As far as we are aware this is also (independently) presented in the 1993 work by Kaliski Jr. [18].

change the modular multiplication accordingly. This allows replacing the typically required division with a multiplication. In practice this is often faster by a constant factor. Since a change of representation of the input and the output is required Montgomery reduction is best used when a long series of modular arithmetic is needed; a setting which is common in public-key cryptography.

Given a modulus q co-prime to r, the idea is to select the Montgomery radix r^n such that $r^{n-1} < q < r^n$. Typically one chooses r to be 2^w. Montgomery reduction requires a pre-computed constant $\mu = -q^{-1} \bmod r^n$ which depends on the modulus q used. Given an integer c (such that $0 \leq c < q^2$) Montgomery reduction computes

$$\frac{c + (\mu \cdot c \bmod r^n) \cdot q}{r^n} \equiv c \cdot r^{-n} \pmod{q}. \tag{1}$$

It should be noted that the division by r^n is an exact division and in practice can be computed by a simple right shift when r is a power-of-two. Similarly, $\mu \cdot c \bmod r^n$ in Eq. (1) can be computed efficiently by computing only the n least significant computer words.

In order to use Eq. (1) one needs to change the representation of $a, b \in \mathbf{Z}_q$ to $\tilde{a} = a \cdot r^n \bmod q$ and $\tilde{b} = b \cdot r^n \bmod q$. Then the Montgomery reduction of $\tilde{a} \cdot \tilde{b} \equiv a \cdot b \cdot r^{2n} \pmod{q}$ becomes $a \cdot b \cdot r^{2n} \cdot r^{-n} \equiv a \cdot b \cdot r^n \pmod{q}$ which is the Montgomery representation of $a \cdot b \bmod q$.

Avoiding the conditional subtraction. Whenever $0 \leq c < q^2$ then

$$0 \leq \frac{c + (\mu \cdot c \bmod r^n) \cdot q}{r^n} < 2q$$

and a single conditional subtraction of q is needed to reduce the result to $[0, 1, \ldots, q-1]$. This conditional subtraction can be omitted when the Montgomery radix is selected such that $4q < r^n$ and a redundant representation is used for the input and output values of the algorithm. More specifically, whenever $a, b \in \mathbf{Z}_{2q}$ (the redundant representation) where $0 \leq a, b < 2q$, then the output $a \cdot b \cdot r^{-n}$ is also upper-bounded by $2q$ and can be reused as input to the Montgomery multiplication without the need for a conditional subtraction [31, 33]. Only at the end of a long series of computation a single conditional subtraction may be necessary, to move from the redundant to a regular representation.

Radix-r Montgomery reduction. The Montgomery reduction as presented in Eq. (1) works directly with multi-precision integers. This has the advantage that asymptotically fast approaches for the multiplication with the modulus q can be used. However, the downside is the size of the intermediate results which are stored in $2n+1$ computer words which can be a significant burden on the available registers and memory on constrained devices.

This can be remedied by using the so-called radix-r Montgomery reduction [12]. It computes the reduction step-by-step, each time reducing with r. This means the precomputed Montgomery constant needs to be adjusted to $\mu = -q^{-1} \bmod r$. The algorithms iterates n times to reduce c from $2n$ to n computer words as follows (we use $r = 2^{32}$)

$$c \leftarrow \frac{c + (\mu \cdot c \bmod r) \cdot q}{r} = \frac{\sum_{i=0}^{m-1} c_i 2^{32i} + (\mu \cdot c_0 \bmod 2^{32}) \cdot q}{2^{32}}$$

for $m = 2n$ to $m = n + 1$. In every of the n iterations we perform a 32-bit times $32n$-bit multiplication. The intermediate results now fit in at most $n+1$ computer words. Hence, the computational cost for this interleaved variant requires $n^2 + n$ multiplication instructions.

1.2 Graphics Processing Units

Originally, Graphics Processing Units (GPUs) have mainly been used as a device for gaming and video processing. Due to the increasing computational requirements of graphics-processing applications, GPUs have become very powerful parallel processors and this incited research interest in computing outside the graphics-community. We focus on the general-purpose GPU computing approach by Nvidia called Compute Unified Device Architecture (CUDA) which facilitates the development of massively-parallel general purpose applications for GPUs (cf. [24]).

We briefly recall some of the basic components of NVIDIA GPUs. Each GPU contains a number of streaming multiprocessors (SMs) and each SM consists of multiple scalar processor cores (SP); their numbers vary by graphics card. The C for CUDA is an extension to the C programming language that employs the massively parallel programming model called *single-instruction multiple-thread*. The programmer defines *kernel functions*, which are compiled for and executed on the SPs of each SM in parallel: each light-weight thread executes the same code, operating on different data. A number of threads are grouped into a *thread block* which is scheduled on a single SM, the threads of which time-share the SPs.

The GPU has a large but relatively slow amount of global memory. Global memory is shared among all threads running on all SMs. On a lower level, threads inside each thread block are executed in groups of 32 called warps. By switching between the different warps, trying to fill the pipeline as much as possible, a high throughput rate can be sustained. When the code executed on the SP contains a conditional data-dependent branch all possibilities, taken by the threads inside this warp, are serially executed (threads which do not follow a certain branch are disabled). After executing these possibilities the threads within this warp continue with the same code execution. For optimal performance it is recommended to avoid multiple execution paths within a single warp.

There has been a significant amount of research how to use this computing power offered by GPUs in cryptology. This includes both cryptography (e.g. [28, 32]) as well as cryptanalysis (cf. [4, 14, 23, 17]).

2 Fast Finite Field Arithmetic in SIDH

The first implementation of SIDH [2] uses Barrett reduction [3] to compute arithmetic modulo $2^x3^y - 1$. The special shape of the modulus is not used in this implementation. However, the authors of [8, 22, 15] do use this special shape of the modulus in their high-performance SIDH implementation.

Recall that in SIDH arithmetic is performed in a field $\mathbf{F}_{q^2} = \mathbf{F}(i)$ for $i^2 = -1$ which in turn translates to arithmetic in $\mathbf{F}_q$. The standard approach is to write $a = a_0 + a_1 i$ and $b = b_0 + b_1 i$, for $a, b \in \mathbf{F}_{q^2}$, then $c = ab = c_0 + c_1 i$ where $c_0 = a_0 b_0 - a_1 b_1$ and $c_1 = a_0 b_1 + a_1 b_0$. The naive approach computes four *interleaved* modular multiplications (where the computation of the multiplication and reduction are combined) or four multiplications and two modular reductions. When using Karatsuba multiplication [19] this can be reduced to three multiplications and two modular reductions or three modular multiplications in the interleaved setting. Hence, when computing modular arithmetic in $\mathbf{F}_{q^2}$ computing the multiplications and modular reductions separately is to be preferred from a performance perspective.

Algorithm 1 Radix-2^{32} subtraction-less Montgomery reduction algorithm in product scanning form. The temporary variable c can at all times be represented by at most three 32-bit computer words.

Input: $\begin{cases} q = \sum_{i=0}^{n-1} q_i 2^{32i}, & \text{odd modulus such that } 2^{32(n-1)} < q < 2^{32n} \text{ and } 4q < 2^{32n} \\ x = \sum_{i=0}^{2n-1} x_i 2^{32i}, & \text{integer to be reduced such that } 0 \le x \le (2q-1)^2 \\ \mu, & \text{Montgomery constant } \mu = -q^{-1} \bmod 2^{32}. \end{cases}$

Output: $z = \sum_{i=0}^{n-1} z_i 2^{32i} = x \cdot 2^{-32n} \bmod q$ such that $0 \le z < 2q$.

```
 1: $c \leftarrow 0$
 2: for $i$ from 0 to $n-1$ do
 3:     for $j$ from 0 to $i-1$ do
 4:         $c \leftarrow c + z_j \cdot q_{i-j}$
 5:     $c \leftarrow c + x_i$
 6:     $z_i \leftarrow c \cdot \mu \bmod 2^{32}$
 7:     $c \leftarrow c + z_i \cdot q_0$
 8:     $c \leftarrow \lfloor c/2^{32} \rfloor$
 9: for $i$ from $n$ to $2n-2$ do
10:     for $j$ from $i-n+1$ to $n-1$ do
11:         $c \leftarrow c + z_j \cdot q_{i-j}$
12:     $c \leftarrow c + x_i$
13:     $z_{i-n} \leftarrow c \bmod 2^{32}$
14:     $c \leftarrow \lfloor c/2^{32} \rfloor$
15: $z_{n-1} \leftarrow c + x_{2n-1} \bmod 2^{32}$
```

All recent SIDH implementations use a modified version of Montgomery reduction to compute the modular reduction. As summarized in [5], when using a multiplication instruction which multiplies $wn \times wn$ to $2wn$-bit integers, Montgomery reduction can compute $\mu \cdot c \bmod r^n$ using n multiplication instructions and the multiplication with the modulus q using n^2 multiplication instructions when using a modulus $q = 2^x 3^y - 1 < 2^{wn}$. This can be optimized as follows. Since $q \equiv -1 \bmod 2^w$ it follows that $\mu = -q^{-1} \equiv 1 \bmod 2^w$ and the multiplication by μ becomes negligible. The multiplication by q can be optimized by only using multiplication instructions to compute the product with the 3^y part, the multiplication with 2^x is done using shifts or by reordering computer words in memory. Hence, the $n^2 + n$ multiplication instructions for the computation of the Montgomery reduction can be reduced to $n^2/2$ in the setting of SIDH. For example, reduction modulo $q = 2^{372}3^{239} - 1$, as used in SIKEp761 can be computed as

$$\frac{c + (\mu \cdot c \bmod r)q}{r} = \frac{c + (c \bmod 2^{64})(2^{372}3^{239} - 1)}{2^{64}} =$$
$$c_0 \cdot (2^{308}3^{239}) + \sum_{i=1}^{23-j} c_i 2^{64(i-1)} = 2^{256}2^{52}3^{239} \cdot c_0 + \sum_{i=1}^{23-j} c_i 2^{64(i-1)}.$$

This process is repeated 12 times (for $j = 0$ to 11) and the input c is overwritten as the output for the next iteration. Hence, since $2^{5 \cdot 64} < 3^{239} < 2^{6 \cdot 64}$ (but $2^{52}3^{239} > 2^{6 \cdot 64}$) this approach requires either $12 \cdot 6 = 72$ multiplication instructions together with a shift by 52 bits every iteration or $12 \cdot 7 = 84$ multiplication instructions. The latter approach is being used by the assembly implementation in SIKE.

This was the motivation of Bos and Friedberger in [5] to look for SIDH moduli which allow faster implementation by lifting the restriction of $p = 3$ in the definition of the SIDH modulus. Computing Montgomery reduction modulo their suggested prime: $2^{391}19^{88} - 1$ can

Algorithm 2 Montgomery reduction for moduli of the form $2^{384} \cdot m - 1$ using a radix-$2^{12 \cdot 32}$ approach.

Input: $\begin{cases} q = \sum_{i=0}^{23} q_i 2^{32i}, \text{ odd modulus such that } 2^{32 \cdot 23} < q < 2^{32 \cdot 24} \text{ and } 4q < 2^{32 \cdot 24} \\ c = \sum_{i=0}^{47} c_i 2^{32i}, \text{ integer to be reduced such that } 0 \leq x \leq (2q-1)^2 \end{cases}$

Output: $z = \sum_{i=0}^{23} z_i 2^{32i} = c \cdot 2^{-32 \cdot 24} \bmod q$ such that $0 \leq z < 2q$.

```
1: $C \leftarrow 0$
2: mul($C_1, C_0, c_0, q_0$)
3: for $i$ from 1 to 11 do
4:     muladd($C_{i+1}, C_i, c_0, q_i$)
5: for $i$ from 1 to 11 do
6:     $t \leftarrow 0$
7:     muladd($t, C_i, c_i, q_0$)
8:     for $j$ from 1 to 11 do
9:         muladdadd($t, C_{i+j}, c_i, q_j$)
10:    $C_{i+12} \leftarrow t$
11: adds($d_0, c_{12}, C_0$)
12: for $i$ from 1 to 23 do
13:     adcs($d_i, c_{12+i}, C_i$)
14: for $i$ from 24 to 35 do
15:     adcs($d_i, c_{12+i}, 0$)
16: $C \leftarrow 0$
17: mul($C_1, C_0, d_0, q_0$)
18: for $i$ from 1 to 11 do
19:     muladd($C_{i+1}, C_i, d_0, q_i$)
20: for $i$ from 1 to 11 do
21:     $t \leftarrow 0$
22:     muladd($t, C_i, d_i, q_0$)
23:     for $j$ from 1 to 11 do
24:         muladdadd($t, C_{i+j}, d_i, q_j$)
25:     $C_{i+12} \leftarrow t$
26: adds($z_0, d_{12}, C_0$)
27: for $i$ from 1 to 23 do
28:     adcs($z_i, d_{12+i}, C_i$)
```

be done more efficiently. It avoids these additional multiplications or shifts since

$$\frac{c + (\mu \cdot c \bmod r)q}{r} = \frac{c + (c \bmod 2^{64})(2^{391}19^{88} - 1)}{2^{64}} =$$
$$c_0(2^{327}19^{88}) + \sum_{i=1}^{23-j} c_i 2^{64(i-1)} = 2^{320} c_0 (2^7 19^{88}) + \sum_{i=1}^{23-j} c_i 2^{64(i-1)}.$$

(again for for $j = 0$ to 11). Now both quantities 19^{88} and $2^7 19^{88}$ are between $2^{5 \cdot 64}$ and $2^{6 \cdot 64}$. A comparison of run-times shows that this results in almost a 12% speed-up in the modular reduction routine [5].

The implementations of SIKE [1] implement the Montgomery reduction algorithm in product scanning form. It is given, without SIDH specific optimizations in Algorithm 1.

3 Faster Finite Field Arithmetic in SIDH

In this section we study two approaches in order to achieve faster arithmetic in $\mathbf{F}_{q^2}$. First, in Section 3.1, we investigate if Montgomery reduction can be computed more efficiently on modern embedded ARM architectures. Second, we study if we can generate moduli which have a more favorable shape for practical implementations in Section 3.2.

3.1 Using the Multiply Accumulate Accumulate Long Instruction

Assume the Montgomery reduction approach is used with SIDH optimizations as described in Section 2. One approach is to make use of the product scanning technique to get a fast implementation which attempts to minimize register usage as done in the various platform specific implementations from [1]. However, on the popular ARM platforms, the multiply-and-accumulate instruction `UMLAL RdLo, RdHi, Rn, Rm` can be used which computes `RdHi` $\cdot$

$2^{32} + \texttt{RdLo} \leftarrow \texttt{Rn} \cdot \texttt{Rm} + \texttt{RdHi} \cdot 2^{32} + \texttt{RdLo}$ where the operands are 32-bit computer words interpreted as unsigned integers. Moreover, starting from ARMv6 and above, there is an Unsigned Multiply Accumulate Accumulate Long instruction `UMAAL RdLo, RdHi, Rn, Rm` which computes $\texttt{RdHi} \cdot 2^{32} + \texttt{RdLo} \leftarrow \texttt{Rn} \cdot \texttt{Rm} + \texttt{RdHi} + \texttt{RdLo}$. This does not result in an overflow since $(2^{32}-1)^2 + 2(2^{32}-1) < 2^{64}$.

Such multiply-and-accumulate instructions are not present on the `x86` platform except for vector instructions working with floating-point values. Motivated by these efficient arithmetic instructions we explore if their usage can result in faster SIDH-specific field arithmetic. For ease of explanation lets assume we have an SIDH modulus $q = 2^{384} \cdot m - 1$ where m is 382 bits and chosen such that it can be used in the SIKE protocol. The presented bit lengths can be trivially adapted to any size. Recall from Section 2 that in every round $j = 0$ to $j = 23$ the input $c < (2q-1)^2 < 2^{48 \cdot 12}$ is reduced as follows

$$\begin{aligned} c = (c + (\mu c \bmod 2^{32})q)/2^{32} \qquad &= \left(\sum_{i=0}^{47-j} c_i 2^{32i} + c_0(2^{384}m - 1)\right)/2^{32} \\ = \left(\sum_{i=1}^{47-j} c_i 2^{32i} + c_0 2^{384} m\right)/2^{32} &= \sum_{i=1}^{47-j} c_i 2^{32(i-1)} + \sum_{i=0}^{12} d_i 2^{32(i+11)} \end{aligned}$$

for $d = c_0 m = \sum_{i=0}^{12} d_i 2^{32i}$. Observe that the first 12 iterations do not update the values $c_0, \ldots, c_{11}$ with the sum $\sum_{i=0}^{12} d_i 2^{32(i+11)}$ since the low words are zero. Hence, one could instead compute this with two larger iterations (instead of the 24 smaller ones) which merges 12 smaller iterations each as

$$\begin{aligned} c &= \sum_{i=12}^{47} c_i 2^{32(i-12)} + \sum_{i=0}^{11} c_i m 2^{32i} \\ c &= \sum_{i=12}^{35} c_i 2^{32(i-12)} + \sum_{i=0}^{11} c_i m 2^{32i}. \end{aligned}$$

Another way to look at this is computing the Montgomery reduction using a larger radix of $2^{12 \cdot 32}$ instead of the usual radix-2^{32} approach.

This has the advantage that the sum $\sum_{i=0}^{11} c_i m 2^{32i}$ can be computed first before adding it to the shifted value of c. The computation of the one-word times twelve word multiplication $c_i m$ and summing these together can be done efficiently using multiply accumulate accumulate long instructions available on modern ARM architectures. Moreover, this *lowers* the total number of additions required for the computation of the Montgomery reduction compared to the product scanning approach with SIDH optimizations. The disadvantage of this technique is that intermediate results are larger and therefore more registers / memory, and instructions to move data from and to memory, are needed to compute the reduction.

Algorithm 2 outlines this approach where the usage of `UMLAL` is denoted by muladd(RdHi, RdLo, Rn, Rm)) and the usage of `UMAAL` is denoted by muladdadd(RdHi, RdLo, Rn, Rm). Addition of values $c = a+b$ with set the carry-out are denoted by adds(c, a, b) and adcs(c, a, b) where the latter also adds the value of the carry-flag as additional input.

3.2 SIDH Moduli with Zeros

In Montgomery reduction a multiplication with the modulus is required. In SIDH one can already speed-up the operation by exploiting the special shape of $q = 2^x m - 1$ where $m = 3^y$

in SIKE. This can be made even faster if one or more of the digits in m are zero. We briefly describe two approaches to generate such moduli.

Generate values of a special shape which are smooth. One approach is to construct many moduli of a particularly "good" shape which allow for even faster reduction for a selected method and checking if they are smooth. For example, moduli $2^x \cdot m - 1$ where almost all of the computer words of m are zero could be considered to speed-up the computation of Montgomery reduction since the multiplication by m can be sped-up by omitting the computations with these zero computer words. In order to estimate how many of such values need to be tested we apply the Dickman–de Bruijn ρ function [11, 9]. This function can be used to estimate the proportion of smooth numbers up to a given bound. More precisely, the number of $x^{1/a}$-smooth numbers (integers with all prime factors below $x^{1/a}$) below x can be estimated by

$$\Psi(x, x^{1/a}) \approx x\rho(a)$$

where it has been shown that this estimate is accurate up to an error of $\mathcal{O}(x/\log x)$ [29]. Hence, in order to estimate the probability that a uniform random number between 2^{348} and 2^{384} (a range typical for moduli used in SIKE [1]) is 4096-smooth is

$$\frac{\Psi(2^{384}, 2^{12}) - \Psi(2^{348}, 2^{12})}{2^{384} - 2^{348}} \approx \frac{2^{384}\rho(32) - 2^{348}\rho(29)}{2^{384} - 2^{348}} \approx 2^{-179}.$$

The value 4096 is chosen somewhat arbitrary but can be considered quite large in the light of the more expensive isogeny computations and the values which are used in our results in Section 5. This clearly shows that attempting to find better moduli with this approach is not practical.

Generate smooth values which have a nice shape. This approaches the problem the other way around. We generate integers as the product of small odd prime powers making them smooth for some selected bound. This limits the negative performance impact on the isogeny computation. At some point some of these generated values will have additional desirable properties, making the modular reduction more efficient. Our main criterion is that one of the digits of the generated smooth value is zero since this means that multiplication by this word together with the associated additions can be omitted. If we generate $32n$-bit integers in such a way then a very crude estimate is that we expect to find such a zero-digit integer once every $2^{32}/(n-2)$ integers since the most-significant and the least-significant computer words cannot be zero. This approach is practical but the performance impact is obviously more limited. It will be investigated further in the upcoming sections.

4 Generating smooth values

As outlined in the previous Section the idea is to generate smooth values m of an appropriate bit-length such that they can be used in SIDH and with the additional requirement that at least one computer word is equal to zero. The SIKE submission specifies three parameter sets targeting three different security levels [1]. The sizes of the moduli used are as follows

Scheme	e_2	e_3	$\lceil \log_2(2^{e_2} \cdot 3^{e_3} - 1) \rceil$	quantum security
SIKEp503	250	159	503	84-bit
SIKEp761	372	239	761	125-bit
SIKEp964	486	301	964	160-bit

Table 1. The number of possible values x which are the product of small prime powers with integer exponents $a > 1$ and $b, c, d, e, f, g \geq 0$ such that $2^{352} < x < 2^{480}$.

x	#candidates	#integers with at least one zero-digit
$5^b \cdot 3^c$	14 406	1 259
$7^a \cdot 5^b \cdot 3^c$	1 072 157	
$11^a \cdot 7^b \cdot 5^c \cdot 3^d$	43 976 602	
$13^a \cdot 11^b \cdot 7^c \cdot 5^d \cdot 3^e$	1 286 638 341	
$17^a \cdot 13^b \cdot 11^c \cdot 7^d \cdot 5^e \cdot 3^f$	27 666 549 374	
$19^a \cdot 17^b \cdot 13^c \cdot 11^d \cdot 7^e \cdot 5^f \cdot 3^g$	485 711 473 846	
$23^a \cdot 19^b \cdot 17^c \cdot 13^d \cdot 11^e \cdot 7^f \cdot 5^g \cdot 3^h$	6 966 901 075 661	21 183
$29^a \cdot 23^b \cdot 19^c \cdot 17^d \cdot 13^e \cdot 11^f \cdot 7^g \cdot 5^h \cdot 3^i$	82 736 661 003 058	180 850

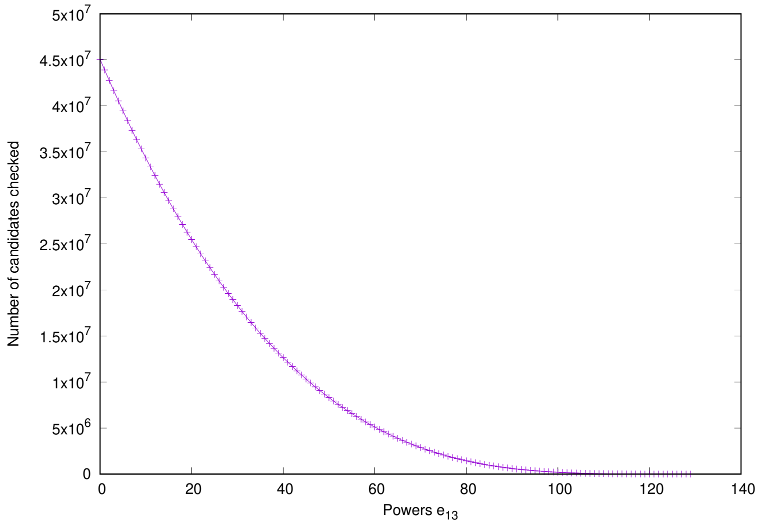

Fig. 1. Plot which shows the number of candidates which need to be inspected for different exponents e_{13} in $x = 13^{e_{13}} \cdot 11^b \cdot 7^c \cdot 5^d \cdot 3^e$ such that $2^{352} < x < 2^{480}$.

We target the two higher security levels and look for values of m such that $11 \cdot 32 < m < 15 \cdot 32$. Since 2^e and m have approximately the same bit-length due to security considerations this gives a prime in the range of SIKEp761 and SIKEp964.

We implemented a brute-force search algorithm which for the first s odd primes below some bound B generates all possible B-smooth integers in this specified range and tests if one of the 32-bit computer words is zero. We show incremental results in Table 1 for $(s, B) = (2, 5)$ until $(s, B) = (9, 29)$ together with how many integers were checked and how many zero-digit integers were found. This approach is already quite successful, we managed to identify over 200 000 unique integers which are 29-smooth and have a zero-digit, we will see in the next subsection that this does not seem to lend itself directly to run concurrently on devices which allow massive parallelism.

4.1 GPU Implementation

The approach from the previous section simply iterates over all possible prime powers for one prime-base at-a-time. Using the notation from Table 1, the run-time for $a = 1$ is significantly longer than for $a = 50$. This is due to the remaining search space being much larger since

Algorithm 3 Brute-force GPU algorithm to search for smooth moduli with a zero 32-bit computer word.

```
Input: {
  $t \in \mathbf{Z}$,     thread index
  $\ell \in \mathbf{Z}$,     size of the set $S_2$
  $S_2$,        set of $\ell$ odd primes
  $L$,          list of input values
  count,      value updated by *all* threads which keeps track of how
              many integers have been found (initial value is zero).
Output: The list $\text{output}_i$, for $0 \le i <$ count, of integer values such that $2^{32\cdot 11} < \text{output}_i < 2^{32\cdot 16}$ and at least
  one of the 32-bit computer words is zero.
 1: for $j$ from 0 to $\ell - 1$ do
 2:     $c_j \leftarrow L_t$
 3: while true do
 4:     $x \leftarrow c_{\ell-1}$
 5:     if $2^{32\cdot 11} < x = \sum_{i=0}^{n} x_i 2^{32i} < 2^{32\cdot 16}$ and $x_j = 0$ for one or more $0 \le j \le n$ then
 6:         oldCount $\leftarrow$ atomicAdd(count, 1)
 7:         $\text{output}_{\text{oldCount}} \leftarrow x$
 8:     $x \leftarrow x \times s_{\ell-1}$
 9:     if $x \ge 2^{32\cdot 16}$ then
10:         $k \leftarrow \ell - 1$
11:         do
12:             if $k = 0$ then
13:                 return
14:             $k \leftarrow k - 1$
15:             $c_k \leftarrow c_k \cdot s_k$
16:         while $c_k \ge 2^{32\cdot 16}$
17:     for $j$ from $k + 1$ to $\ell - 1$ do
18:         $c_j \leftarrow c_k$
```

the starting value is lower. Accelerating the search by running this on a GPU seems like a natural fit: we could generate and check many integers in parallel without any communication between the different threads.

However, the naive approach of assigning every thread a different power of the first prime base seems inefficient. Consider the setting where five primes are used. Figure 1 shows the number of integer candidates which need to be checked per exponent of the largest prime base 13. The threads which get assigned the smallest exponents do almost all the work of the entire computation. In the GPU setting this means that many threads will stall and wait for the computations of the threads assigned with these small exponents since they all need to follow the exact same computational steps.

Therefore, we decided to go for a different strategy for the GPU implementation. In order for every thread to compute roughly the same amount of work we select two sets S_1 and S_2. The set $S_1 = \{p_0, \ldots, p_{k-1}\}$ consists of k prime values p_i used in a pre-computation phase while $S_2 = \{s_0, \ldots, s_{\ell-1}\}$ consists of ℓ prime values s_j for the online phase such that $S_1 \cap S_2 = \emptyset$. For a given bound $\mathfrak{B}$, the precomputation phase computes all the possible products $\prod_{i=0}^{k-1} p_i^{e_i} < \mathfrak{B}$ for $p_i \in S_1$ and enumerating all possible positive integer exponent values e_i. This large list of integers is used as input to the GPU. Every thread picks a number from this list and enumerates all possible exponents of $s_j \in S_2$ such that $2^{352} < \prod_{i=0}^{k-1} p_i^{e_i} \cdot \prod_{j=0}^{\ell-1} s_j^{\hat{e}_j} < 2^{480}$. When k is not too small, most pre-computed values $\prod_{i=0}^{k-1} p_i^{e_i}$ will be of approximately the size $\mathfrak{B}$. Hence, the amount of work per thread is similar: this is further improved by sorting the precomputed list by size. This overcomes the problem outlined in

Table 2. Overview of the number of different ARM assembly instructions used to implement the different described approaches. The value of m is assumed to be less than 2^{382}.

algorithm	prime	zero word	mul `umull`	mul `umlal`	mul `umaal`	add	ldr,str,mov	eor
product scanning	$2^{372}3^{239}-1$	×	1	311	0	731	1101	347
product scanning	$2^{384}m-1$	×	1	287	0	677	1027	322
		✓	1	263	0	627	955	298
radix-2^{384}	$2^{384}m-1$	×	2	44	242	60	1344	44
		✓	2	42	220	104	1296	66

Figure 1 while it still allows to efficiently check large ranges of smooth numbers by choosing the sets S_1 and S_2 suitably.

Algorithm 3 shows the part executed by the GPU. The function `atomicAdd` is a GPU-specific function which allows a thread to read-modify-write a value to a global memory location. It is atomic in the sense that it ensures that no other threads interfere during the operation. We use this function to determine the correct location in the output list.

5 Performance and Search Results

5.1 Montgomery Reduction on Embedded Platforms

In order to test which arithmetic approach is better; the product scanning approach as used in the SIKE implementation or the radix-2^{384} approach presented in Section 3.1, we implemented both algorithms in assembly for the BeagleBone Black. This is a development platform featuring a 32-bit AM335x 1GHz ARM Cortex-A8. Since the SIKE submission package [1] only has optimized assembly code for the ARM64 and x64 platforms we ported the product scanning approach to this ARM platform. The approach is identical to the one from the SIKE authors and the only thing we changed was to merge the multiply and addition instruction into the dedicated multiply-and-add (`umlal`) instruction.

In order to give an impression how these assembly implementations of the different algorithms compare to each other we give an overview of the number of instructions used in Table 2. The first row shows the results of the product scanning implementation using the prime from SIKE directly ported to the 32-bit ARM. The second row shows this for the more efficient primes given in [5], where the m part is chosen small enough to fit in fewer computer words. Just as observed in [5] for the x86 platform this reduces the number of multiplication instructions needed on the ARM architecture. We also present the instruction count for m chosen such that one 32-bit computer word is zero. The last row in Table 2 shows the radix-2^{384} approach. By using the unsigned multiply accumulate accumulate long instruction `umaal` the number of addition instructions are reduced by more than one order of magnitude. However, as expected, the total number of load and store instructions is higher than for the product scanning approach. The exact benchmark results for this approach are reported in Section 5.3.

5.2 Searching for Faster SIDH-Moduli

We have implemented and executed Algorithm 3 on a Nvidia Quadro K5000 GPU. This GPU has 1536 CUDA cores, 4GB of global memory and the core clock runs at 706MHz. The CPU we used for testing is an Intel Xeon CPU E5-2650 (running at 2.60GHz): this machine has 8 CPU cores. After doing some basic throughput experiments we found that running 32 blocks of 256 threads each optimize throughput for this GPU and for the size of numbers considered in this work. All of the 8192 threads are assigned a 64-bit value: hence, we use a value of $\mathfrak{B} = 2^{64}$ in the notation of Section 4.1. Moreover, we use $S_1 = \{17, 19, 23, 29, 31, 37, 41, 43, 47, 53\}$ and $S_2 = \{3, 5, 7, 11, 13\}$.

The GPU significantly outperforms the CPU. Given an integer $\prod_{i=0}^{k-1} p_i^{e_i} < 2^{64}$, where the $p_i \in S_1$, one has to inspect approximately $2^{29.5}$ candidates to check if one of the 32-bit computer words is zero. The single-core CPU implementation executes in 167 seconds. Since there are 8 cores this means 20.8 seconds per number of the entire CPU. The GPU implementation has a significantly longer latency: 11585 seconds which is over 3.2 hours compared to the latency of less than 3 minutes on the CPU. However, the GPU can execute 8192 in parallel. This means that the throughput is 1.4 seconds per input number on this GPU: almost a factor 15 improvement.

Given the sets S_1 and S_2 as defined above we generated over $1.1 \cdot 10^6$ 64-bit input integers. Hence, the GPU checked in total around $2^{49.6}$ integers if they have one or more zero-valued 32-bit computer words. As a result we found over $2.3 \cdot 10^6 \approx 2^{21.2}$ such values.

5.3 Performance Results of the Faster SIDH-Moduli

The CPU implementation, as described in Section 4, found over 200 000 unique integers which are 29-smooth and have a zero-digit (see Table 1). Moreover, the GPU implementation found an order of magnitude more integers (around $2^{21.2}$) such that m is 53-smooth. Given this large list of values for m we need to check if the potential SIDH moduli $\hat{q} = 2^x \cdot m - 1$ fulfill a number of requirements in order for them to be usable in practice. The bit-length $\lceil \log_2(m) \rceil$ of m determines the possible values of the exponent x one can use: we select $x \in \{\lceil \log_2(m) \rceil - 10, \ldots, 0, \ldots, \lceil \log_2(m) \rceil + 10\}$ since SIDH security requires $2^x \approx m$ [1]. Of course we also require that $\hat{q}$ is prime. Given the 2 575 077 input values this results in 530 612 pairs of (x, m) such that these requirements are satisfied.

Since this list is still large enough we set additional requirements which are beneficial for a real implementation. First, we require that $x \equiv 0 \bmod 32$ to ensure that the multiplication with the 2^x part is completely for free instead of requiring shift instructions. The subtraction-less variant of Montgomery reduction as described in Section 1.1 additionally requires that $\lceil \log_2(m) \rceil \bmod 32 < 30$ and $\lceil \log_2(m) \rceil < x$ to ensure $4\hat{q} < 2^{2x}$.

Even with these additional restrictions, there are 787 result pairs $(384, m)$ when $x = 384$. When we select pairs with the minimum bit-difference between 384 and $\lceil \log_2(m) \rceil$ 102 pairs are left. Similarly, when $x = 480$ this results in 2535 pairs $(480, m)$ which fulfill these requirements and 370 of them are left when the bit-difference is minimal. As an example we show two of the values found below where we looked for values of m which maximize the power of 3 and additionally preferring m which have smaller prime divisors: both for performance considerations when computing the large odd-degree isogenies. For the prime $2^{384} \cdot m - 1$, which could be considered as a replacement in SIKEp761, one can use

$$\begin{aligned} m = {} & 3^{154} \cdot 5^5 \cdot 7^{22} \cdot 11^6 \cdot 17^3 \cdot 29^3 \cdot 37^2 \cdot 43^1 \\ & \texttt{0x1dba73ea e32a1380 d3733fb7 ececcb64} \\ = {} & \texttt{565dcf70 a6b12bb5 7da00322 3a32f0cb} \\ & \texttt{56bf9bb6 00000000 fa1d470f f92bce7b}. \end{aligned}$$

Table 3. Benchmark results summary of the various implementation strategies for Montgomery reduction modulo the modulus given where $m < 2^{382}$ on the BeagleBone Black. The zero-digit column states if the reduction assumes one of the computer words of m is zero. The column with $\bar{x} \pm \sigma$ gives the mean $\bar{x}$ and standard deviation σ expressed in the number of cycles.

algorithm	prime	zero word	$\bar{x} \pm \sigma$
product scanning [1]	$2^{372}3^{239} - 1$	$\times$	3738 ± 11
product scanning	$2^{384}m - 1$	$\times$	3473 ± 55
		$\checkmark$	3202 ± 12
radix-2^{384}	$2^{384}m - 1$	$\times$	2595 ± 12
		$\checkmark$	2492 ± 9

Similarly, for SIKEp964 one could use $2^{480} \cdot m - 1$ where

$$m = 3^{192} \cdot 5^{17} \cdot 7^{9} \cdot 11^{4} \cdot 13^{10} \cdot 17^{5} \cdot 19^{1} \cdot 31^{2} \cdot 43^{1} \cdot 47^{3}$$
$$= \begin{array}{r} \texttt{0x17e08fbd 62bb1dd4 00000000 32a79450} \\ \texttt{aa7ac081 f870e00d 03c471eb 0826bd2c} \\ \texttt{d2a4dfca 584e9ffb ae0b6729 8400ee97} \\ \texttt{113129cc 7c343b7c cf233d65.} \phantom{\texttt{00000000}} \end{array}$$

We implemented the product scanning approach as used in [1] as well as the radix-2^{384} approach for the 32-bit ARM platform. The benchmark figures presented in this Section are obtained by accessing the Cycle CouNT (CCNT) Register. We measure the time to compute 10^3 dependent Montgomery reductions and store the mean cycle count for a single operation. This process is repeated 10^3 times and from this set the mean $\bar{x}$ and standard deviation σ are computed. We remove the outliers (more than 2.5 standard deviations away from the mean) and these figures are reported.

Similar to [5] we compare the Montgomery reduction for the prime $2^{372}3^{239} - 1$ from SIKE [1] to an SIDH modulus of the form $2^{384}m - 1$ where $m < 2^{382}$. As summarized in Table 3 the usage of such generic faster primes results in a speedup of a factor of 1.08. This is less than the 12 percent speedup as observed in [5] for 64-bit platforms when $m = 2^{7}19^{88}$ is considered. This is explained by the increased number of multiplications on 32-bit platforms which makes saving one multiplication less significant. When one out of twelve of the 32-bit computer words needed to represent m is zero the observed speed-up is a factor 1.08: this is as one would expect.

As summarized in Table 2 the radix-2^{384} approach from Section 3 significantly reduces the number of addition instructions required due to the usage of the unsigned multiply accumulate accumulate long instructions. When comparing the product scanning to this radix-2^{384} approach the speedup of the latter over the former is significant: a factor of 1.34. The speed-up when one of the computer words of m is zero is a factor of 1.04. This is lower than expected and lower than for the product scanning approach; which is explained by the fact that a zero-computer word does not remove a number of addition instructions but changes a multiply-accumulate-accumulate to a multiply-accumulate.

Overall, when combining the faster radix-2^{384} Montgomery reduction approach and the zero-computer word modulus this results in a reduction which is 1.5 times faster than used in SIKE. This is a promising speed-up which can facilitate the realization of isogeny-based cryptographic protocols on embedded platforms.

6 Conclusions and Future Work

In this work we studied two optimizations to realize faster modular arithmetic for cryptographic schemes based on supersingular isogenies such as SIKE [1] where the focus is on embedded devices like 32-bit ARM platforms. We have shown that a large radix approach for Montgomery reduction can make use of the special fused multiply-and-accumulate-accumulate instructions and significantly outperforms the product scanning approach currently used in SIKE. Moreover, motivated by the results from [7], which shows new efficient formulas for computing arbitrary odd-degree isogenies between elliptic curves in Montgomery form, we search for faster SIDH-friendly moduli. Using the computational power of GPUs we manage to find a large set of such moduli for different security levels where one 32-bit computer word is zero. When combining these two optimizations the performance of the modular reduction is increased by a factor 1.5.

It remains an interesting question if such larger odd-degree isogenies have a negative impact on the security. Moreover, it would be even more beneficial if we could find multiple 32-bit computer words which are zero. Even better would be a 64-bit computer word equal to zero such that high-end servers can also benefit from this arithmetic speed-up.

Acknowledgments

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation program Marie Skłodowska-Curie ITN ECRYPT-NET (Project Reference 643161) and Horizon 2020 project PQCRYPTO (Project Reference 645622).

References

1. 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 Post-Quantum Standardization project, 2017.
2. R. Azarderakhsh, D. Fishbein, and D. Jao. Efficient implementations of a quantum-resistant key-exchange protocol on embedded systems. Technical report, `http://cacr.uwaterloo.ca/techreports/2014/cacr2014-20.pdf`, 2014.
3. P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Heidelberg, Aug. 1987.
4. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In A. Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 483–501. Springer, Heidelberg, Apr. 2009.
5. J. W. Bos and S. Friedberger. Fast arithmetic modulo $2^x p^y \pm 1$. In N. Burgess, J. D. Bruguera, and F. de Dinechin, editors, *IEEE Symposium on Computer Arithmetic – ARITH 2017*, pages 148–155. IEEE Computer Society, 2017.
6. L. Chen, S. Jordan, Y. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. NISTIR 8105, National Institute of Standards and Technology, 2016. `http://csrc.nist.gov/publications/drafts/nistir-8105/nistir_8105_draft.pdf`.
7. C. Costello and H. Hisil. A simple and compact algorithm for SIDH with arbitrary degree isogenies. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 303–329. Springer, Heidelberg, Dec. 2017.
8. C. Costello, P. Longa, and M. Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 572–601. Springer, Heidelberg, Aug. 2016.
9. N. G. De Bruijn. On the number of positive integers $\leq x$ and free of prime factors $> y$. In *Proc. Kon. Ned. Akad. Wetensch. Ser. A*, volume 54, pages 50–60, 1951.

10. M. H. Devoret and R. J. Schoelkopf. Superconducting circuits for quantum information: an outlook. *Science*, 339(6124):1169–1174, 2013.
11. K. Dickman. On the frequency of numbers containing prime factors of a certain relative magnitude. *Arkiv for matematik, astronomi och fysik*, 22(10):1–14, 1930.
12. S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 230–244. Springer, Heidelberg, May 1991.
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*, 2017.
14. J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel. Parallel shortest lattice vector enumeration on graphics cards. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT 10*, volume 6055 of *LNCS*, pages 52–68. Springer, Heidelberg, May 2010.
15. A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao. Supersingular isogeny Diffie-Hellman key exchange on 64-bit arm. *IEEE Transactions on Dependable and Secure Computing*, 2017.
16. D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In B. Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *LNCS*, pages 19–34. Springer, 2011.
17. A. Joux and V. Vitse. A crossbred algorithm for solving boolean polynomial systems. In J. Kaczorowski, J. Pieprzyk, and J. Pomykala, editors, *Number-Theoretic Methods in Cryptology 2017*, volume 10737 of *LNCS*, pages 3–21. Springer, 2018.
18. B. Kaliski Jr. The Z80180 and big-number arithmetic. *Dr. Dobb's Journal*, 18(9):50–58, September 1993.
19. A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akad. Nauk SSSR*, number 145 in Proceedings of the USSR Academy of Science, pages 293–294, 1962.
20. A. Karmakar, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient finite field multiplication for isogeny based post quantum cryptography. In S. Duquesne and S. Petkova-Nikova, editors, *Arithmetic of Finite Fields - WAIFI*, volume 10064 of *LNCS*, pages 193–207, 2016.
21. J. Kelly, R. Barends, A. G. Fowler, A. Megrant, E. Jeffrey, T. C. White, D. Sank, J. Y. Mutus, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, I.-C. Hoi, C. Neill, P. J. J. O/'Malley, C. Quintana, P. Roushan, A. Vainsencher, J. Wenner, A. N. Cleland, and J. M. Martinis. State preservation by repetitive error detection in a superconducting quantum circuit. *Nature*, 519:66–69, 2015.
22. 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 S. Foresti and G. Persiano, editors, *Cryptology and Network Security*, pages 88–103. Springer International Publishing, 2016.
23. P.-C. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. Buchmann, C.-M. Cheng, and B.-Y. Yang. Extreme enumeration on GPU and in clouds - - how many dollars you need to break SVP challenges -. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 176–191. Springer, Heidelberg, Sept. / Oct. 2011.
24. E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
25. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
26. D. Moody. Post-quantum cryptography: NIST's plans for the future. Presentation at PKC 2016, `http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/pqcrypto-2016-presentation.pdf`, 2016.
27. M. Mosca. Cybersecurity in an era with quantum computers: Will we be ready? Cryptology ePrint Archive, Report 2015/1075, 2015. `http://eprint.iacr.org/2015/1075`.
28. A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In S. D. Galbraith, editor, *Proceedings of the 11th IMA international conference on Cryptography and coding*, Cryptography and Coding 2007, pages 364–383. Springer-Verlag, 2007.
29. V. Ramaswami. On the number of positive integers less than x and free of prime divisors greater than x^c. *Bulletin of the AMS*, 55(12):1122–1127, 1949.
30. M. Scott. Fast machine code for modular multiplication. `ftp://ftp.computing.dcu.ie/pub/crypto/fastmodmult2.ps`, 1995.
31. M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In E. E. Swartzlander Jr., M. J. Irwin, and G. A. Jullien, editors, *11th Symposium on Computer Arithmetic*, pages 252–259. IEEE Computer Society, 1993.
32. R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *CHES 2008*, volume 5154 of *LNCS*, pages 79–99. Springer, Heidelberg, Aug. 2008.

33. C. D. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35:1831–1832, 1999.