PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication

Charles. F. Xavier
charlie@ingonyama.com

***Abstract*—Multi-Scalar Multiplication (MSM) is a fundamental computational problem. Interest in this problem was recently prompted by its application to ZK-SNARKs, where it often turns out to be the main computational bottleneck.**

In this paper we set forth a pipelined design for computing MSM. Our design is based on a novel algorithmic approach and hardware-specific optimizations. At the core, we rely on a modular multiplication technique which we deem to be of independent interest.

We implemented and tested our design on FPGA. We highlight the promise of optimized hardware over state-of-the-art GPU-based MSM solver in terms of speed and energy expenditure.

***Index Terms*—Zero-Knowledge, Hardware acceleration, Multi-Scalar Multiplication (MSM), FPGA**

I. Introduction

The multi-product problem is defined as follows: given a sequence of elements $(t_0, t_1, \dots t_{n-1})$, compute the following sequence of products using the minimal number of multiplications possible:

$$y_i = \prod_{j=0}^{n-1} t_j^{x_{i,j}}; \forall i = 0, 1, \dots k-1 \tag{1}$$

Here $x_{i,j} \in \mathbb{Z}_2 : \{0, 1\}$ are elements of a matrix $X_{k \times n}$. When $x_{i,j} \in \mathbb{Z}_r, r \geq 2$ then the sequence y_i computes various exponents of the elements t_j and (1) is known as a multi-exponentiation problem. Four decades ago, Pippenger provided an asymptotically optimal algorithm for both of these problems [1].

Some of the problems that can be reduced to the multi-product or multi-exponentiation are evaluating sparse multivariate polynomials and computing matrix-vector products. The logarithmic version of the same multi-exponentiation problem can be equivalently stated as a Multi-Scalar Multiplication (MSM)

$$\tilde{y}_i = \sum_{j=0}^{n-1} x_{i,j} \tilde{t}_j. \tag{2}$$

One interesting application of the MSM problem is in cryptography, and in particular ZK-SNARKs. In this instance of the problem, we are interested in computing a single product y, where $\tilde{t}_j = G_j$ are points of an Elliptic Curve (EC) group $\mathcal{G}$ over a prime order field $\mathbb{F}_q$ and x_j are scalars s.t. $x_j \in \mathbb{F}_{|\mathcal{G}|}$. MSM as defined above is represented throughout this paper as

$$\text{MSM}(x, G) = \sum_{j=0}^{n-1} x_j G_j \tag{3}$$

Note that at the end of the computation (3) the left-hand side is a single EC point from $\mathcal{G}$.

A. MSM dominates ZK prover computation

A Zero Knowledge Proof (ZKP) is a protocol between two parties, a prover, and a verifier. The prover aims to convince the verifier that a statement is true with high probability, without revealing any additional information. Consider an NP relation $\mathcal{R}(x, w)$ with a corresponding language $\mathcal{L}(\mathcal{R})$, where x is a public input known to both the prover and the verifier and the "witness" w is known only to the prover. For example, think of a boolean circuit with some assignments known to both parties (x) and the other ones known only to the prover (w).

A proof π must possess several properties. At the minimum we require completeness, soundness, and zero-knowledge for which formal definitions can be found in [2]. ZK-SNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) are a subtype of ZK proofs with several additional properties like small proof size and fast verification. This allows ZK-SNARKs to serve as a foundation for highly efficient verifiable computation. The trade-off is in the prover's complexity which becomes a computational bottleneck. See [3] for several references on the subject at various levels, and [4] for a practical technical introduction to the inner workings of ZK-SNARKs.

We use the term *arithmetization* to define the process of "codifying" a relation $\mathcal{R}$ in a set of arithmetic constraints over a finite field $\mathbb{F}$. It is a crucial step, aimed at describing the problem in a unified arithmetic language, enabling the prover to then apply information-theoretic and cryptographic tools for generating the proof. Examples of popular formats in which the "codified" relations are represented are QAP [5], [6] and R1CS.

A necessary step in any ZK-SNARK protocol is efficiently evaluating polynomials at points requested by the verifier. A key cryptographic primitive for an interactive protocol that proves/verifies polynomial relations is a Polynomial Commitment (PC) scheme. PC refers to a computationally binding commitment to a polynomial P, using which the committer can prove the evaluation $P(k)$ at any point $k \in \mathbb{F}$ without revealing P at other points (hiding).

In table I we survey some of the popular commitment schemes used in ZKPs.

We see that for example, Plonk [10] and Marlin [11] use the pairing-based KZG10 [7] scheme. Bulletproofs [12] use

Scheme	KZG10 [7]	IPA [8]	DARK [9]
Cryptographic primitive	Pairing	Discrete Log	Group of unknown order
Ingredients	$\mathcal{G}/\mathbb{F}_q : G_1 \in \mathbb{G}_1, G_2 \in \mathcal{G}_2, G_T \in \mathcal{G}_T$ Pairing: $e(\mathcal{G}_1, \mathcal{G}_2) \rightarrow \mathcal{G}_T$ SRS $= (\tau^0 G_1, \tau^1 G_1, \dots \tau^{n-1} G_1)$	$\mathcal{G}/\mathbb{F}_q : G_i \in \mathcal{G}$ random$(G_i) \leftarrow \mathcal{G}$	$\lvert\mathcal{G}\rvert$ unknown random$(G_i) \leftarrow \mathcal{G}$ $q \in \mathbb{Z}_p; q \sim \mathcal{O}(p \log_2 n)$
Trusted setup	Yes	No	depends on $\mathcal{G}$
Comm(P) $P(X) = \sum_{i=0}^{n-1} a_i X^i$	$P(\tau)G_1 = \sum_{i=0}^{n-1} a_i \text{SRS}_i$	$\langle a, G \rangle = \sum_{i=0}^{n-1} a_i G_i$	$\sum_{i=0}^{n-1} a_i Q_i$ $Q = (q^0 G, q^1 G, q^2 G, \dots q^{n-1} G)$
Computational primitive	MSM	MSM*	MSM*
ZKP's	Plonk [10], Marlin [11]	BulletProofs [12]	Supersonic [9]

TABLE I
CRYPTOGRAPHIC PRIMITIVES TO COMPUTATIONAL PRIMITIVES

inner product arguments [8] that rely on the discrete logarithm assumption. Groups of unknown order play a central role in DARK (Diophantine Argument of Knowledge) [9], used by Supersonic. All of the above protocols rely on MSM to compute polynomial commitments. In table I the MSM* notation is used to highlight that G_i in IPA and multiples of G in DARK are not predetermined and computed during proof generation. On the other hand, in KZG10 the bases $\tau^i \cdot G_1$ are precomputed before proving.

One last classic example of a popular protocol that also requires the prover to compute several MSMs is Groth16 [13].

To sum up, table I highlights the strong presence of MSM in polynomial commitments. While there is sufficient motivation to consider MSM as a standalone problem, even purely from a theoretical standpoint, it also turns out to be a key computational problem that needs to be tackled in order to accelerate ZK-SNARK computation.

In table II, we see that some of the most popular protocols spend a significant fraction of time computing MSMs. While

Protocol	**No. of MSM (Prover)**	**Prover's time** %
Groth16 [13]	4 in $\mathcal{G}_1$, 1 in $\mathcal{G}_2$	$70 - 80\%$
Marlin [11] + Lunar [14]	11 in $\mathcal{G}_1$	$70 - 80\%$
Plonk [10]	9 in $\mathcal{G}_1$	$85 - 90\%$

TABLE II
MSM DOMINATION IN ZKP

the exact percentage of time can vary depending on the implementation and circuit size, in general, MSM is the main computational bottleneck for ZK-SNARK-based proof systems.

In this paper we break down the acceleration of MSM (3) into three sections as follows

- Efficient modular multiplication §II
- Elliptic curve point addition §III
- Multi-Scalar Multiplication §IV

We start with modular multiplication, a fundamental primitive for all finite field arithmetic. There exist efficient modular reduction methods for primes of a special form. However for the general case, which we consider, our modular reduction method needs to work with arbitrary primes. Several such methods can be found in the literature: Barrett reduction [15], Montgomery reduction [16], and lookup table-based methods [17]. In §II we present an optimized variant of the modular multiplication method based on Barrett reduction.

Moving on to the higher level, in §III we discuss the elliptic curve addition based on complete formulae which have been overlooked in the literature, and point out their efficiency in data sampling and parallelism.

Finally, in §IV we discuss our algorithmic optimizations for the efficient parallel implementation of the MSM solver.

Having all the theoretical tools in hand, we implemented our design on Xilinx U55C FPGA. In §V, we demonstrate how our implementation compares to state-of-the-art GPU-based MSM solver.

We conclude with further potential algorithmic improvements in §VI.

B. Related Work

Sppark [18] is a new library developed by Supranational that provides GPU implementation for primitives used in ZKP, among them MSM. We compare our implementation to Sppark in section V.

The best existing FPGA implementation for SNARK are projects led by Ben Devlin for Zcash [19] and the Ethereum Foundation [20]. The projects either implement a limited version of MSM with a naive approach, resulting in poor performance, or skip MSM altogether.

PipeZK [21] is a recent work that provides end-to-end pipelined design for ZK-SNARK. However, PipeZK chose ASIC as its target hardware, skipping over FPGAs. ASIC is at best years away from being accessible while FPGAs are available today with supply time similar to GPUs or even instantly in the cloud [22]. The measurements in PipeZK are therefore done via simulation of ASIC without a physical silicon chip. Our measurements on the other hand are done on physical hardware.

II. EFFICIENT MODULAR MULTIPLICATION

The standard modular multiplication problem in $\mathbb{F}_q$ is formulated as

$$r = a \cdot b \mod q \tag{4}$$

where $a, b, r \in \mathbb{F}_q$, q is a prime. Equivalently this can be written as

$$a \cdot b = l \cdot q + r \tag{5}$$

with $l \in \mathbb{Z}$ such that $0 \le r < q$. In this section we describe an efficient, hardware-friendly method for computing (4). For the rest of this paper assume that all variables are represented in binary, and denote by n the number of digits used to represent any element in $\mathbb{F}_q$, i.e $n = \lceil \log_2 q \rceil$. We begin by recounting the details of the Barrett reduction method.

A. Barrett reduction

Our strategy is to assume that l is approximately known and we denote by $\hat{l}$ the corresponding approximation

$$l - \lambda \le \hat{l} \le l \tag{6}$$

where $\lambda \ge 0$ is a known constant.

1) Case 1: $\lambda = 0$: It is clear by definition (5) that

$$ab[2n-1:0] - \hat{l}q[2n-1:0] = r[n-1:0] \tag{7}$$

where the brackets denote bit sizes. Writing out bits explicitly, (7) is rewritten as:

$$\begin{array}{cccccc} ab[2n-1] & \dots & ab[n] & ab[n-1] & \dots & ab[0] \\ - \ \hat{l}q[2n-1] & \dots & \hat{l}q[n] & \hat{l}q[n-1] & \dots & \hat{l}q[0] \\ \hline 0 & \dots & 0 & r[n-1] & \dots & r[0] \end{array}$$

Note that only $ab[n-1:0]$ and $\hat{l}q[n-1:0]$ are necessary in order to execute the computation. Also note that the result is always a positive n-bit number, so any carry to the $n+1$'st bit is ignored.

2) Case 2: $\lambda > 0$: In this case

$$ab - \hat{l}q = r + \lambda q \tag{8}$$

the number of bits required to represent the right-hand side of the above is

$$\lceil \log_2(r + \lambda q) \rceil \le n + \left\lceil \log_2 \frac{r + \lambda q}{q} \right\rceil \le n + \lceil \log_2(1+\lambda) \rceil \tag{9}$$

so instead of the lowest n bits in (7) we must take $n + \lceil \log_2(1+\lambda) \rceil$. For example, if $\lambda = 3$, the total number of additional bits required would be 2.

3) Approximating l: The only remaining part is to compute the approximation $\hat{l}$:

$$l = \left\lfloor \frac{ab}{q} \right\rfloor = \lim_{k \to \infty} \frac{ab \cdot m(k)}{2^{k+n}} \tag{10}$$

where

$$m(k) = \left\lfloor \frac{2^{k+n}}{q} \right\rfloor < 2^{k+1} \tag{11}$$

For a finite k, the approximation error of $1/q$ is

$$e(k) \equiv \frac{1}{q} - \frac{m(k)}{2^{k+n}} < 2^{-(k+n)} \tag{12}$$

where the upper bound can be derived by examining the maximal difference between the binary representation of the left and right terms. This immediately leads to the approximation error on $l(k)$

$$e(l,k) \equiv \frac{ab}{q} - \frac{ab \cdot m(k)}{2^{k+n}} < 2^{2n} \cdot 2^{-(k+n)} = 2^{n-k} \tag{13}$$

Thus if $k \ge n$ the error is at most 1. Choosing $k = n$ (i.e. $m(n) < 2^{n+1}$) leads to the following approximation on l

$$\hat{l}_0 = \left\lfloor \frac{abm}{2^{2n}} \right\rfloor \ , \ e(\hat{l}_0) < 1 \tag{14}$$

where the approximation error obeys (13).

Let us instead perform the above multiplication in two stages. Initially assume that $ab[2n-1:0]$ is available and perform the multiplication as follows

$$\frac{abm}{2^{2n}} = \frac{ab[2n-1:n] \cdot m}{2^n} + \frac{ab[n-1:0] \cdot m}{2^{2n}} \tag{15}$$

$$< \frac{ab[2n-1:n] \cdot m}{2^n} + 2 \tag{16}$$

This immediately leads to the following approximation on l

$$\hat{l}_1 = \left\lfloor \left\lfloor \frac{ab}{2^n} \right\rfloor \cdot \frac{m}{2^n} \right\rfloor \ , \ e(\hat{l}_1) < 3 \tag{17}$$

where the inner multiplication is n-by-n-bit, the outer multiplication is n-by-$(n+1)$-bit, and the approximation error is upper bounded by the sum of (14) and the right-most term of (16).

In fig 1 we present a block diagram of the modular multiplier. The diagram uses the $\hat{l}_1$ approximation for l given in (17). Note that although multiplication $\lfloor \frac{ab}{2^n} \rfloor \cdot m$ is n-by-$n+1$ bit, the result is less than $q2^n$ so it's $2n$-bit long and so $\hat{l}_1$ is n-bit long.

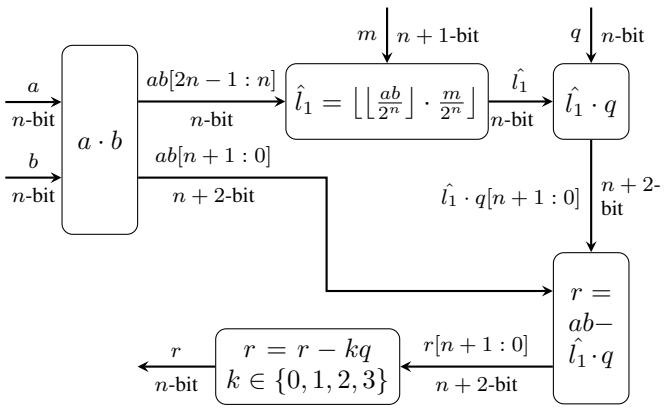

Fig. 1. Optimized Barrett modular multiplier

B. Optimizing integer multiplications in Barrett algorithm

In general, two large integers x, y can be efficiently multiplied using the Karatsuba algorithm (see e.g., [23]). Writing $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$, one reduces the computation of the product $x \cdot y$ to 3 narrower products as shown below

$$x \cdot y = 2^{2k} x_1 y_1 + x_0 y_0 + 2^k((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) \tag{18}$$

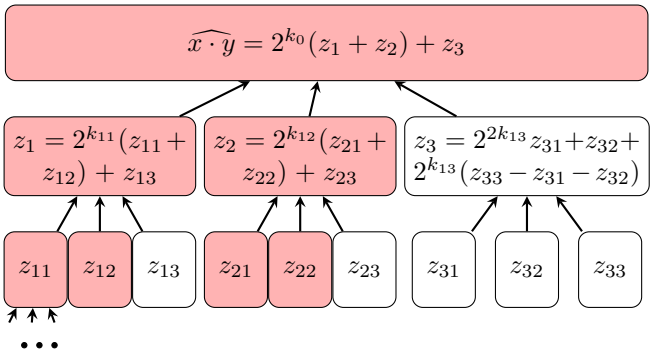

Fig. 2. LSB multiplication

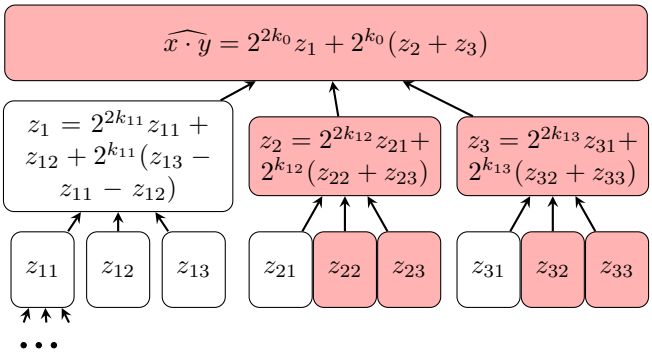

Fig. 3. MSB multiplication

To multiply two large integers, (18) can be iterated before reaching multiplications that are narrow enough to be computed by the hardware directly. However, note that in fig 1, we do not always need the whole result of integer multiplications. In particular, we are only interested in the most significant n bits of $ab[2n-1:n] \cdot m$ and the least significant $n+2$ bits of $l_1 \cdot q$.

We first demonstrate how to compute $n+\sigma$ lowest bits of the product of two n-bit numbers $x \cdot y$ (in the case of algorithm 1, $\sigma = 2$). As before, we write $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$ and expand $x \cdot y = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0$. Note that, as long as $2k \geq n + \sigma$, lowest $n + \sigma$ bits of the product $x \cdot y$ are correctly computed by

$$\widehat{x \cdot y} = 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0 \tag{19}$$

Moreover, we only need the lower $n + \sigma - k$ bits of $x_1 y_0$ and $x_0 y_1$ to be correct. This leads us to a recursive algorithm as shown in fig 2, where the white boxes refer to regular Karatsuba iterations (18) and the pink boxes refer to the simplified iterations (19). To ensure the correctness of the algorithm, we need to make sure that at each simplified iteration (19), k_i satisfies $n + \sigma \leq 2k_i + k_{i-1} + k_{i-2} + \cdots + k_0$.

A more difficult problem is to compute n highest bits of the $2n$-bit product of two numbers $x \cdot y$. Following our earlier approach, we write $x = x_1 2^k + x_0$ and $y = y_1 2^k + y_0$. Since $x \cdot y = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) + x_0 y_0$ we can approximate $x \cdot y$ as

$$\widehat{x \cdot y} = x_1 y_1 2^{2k} + 2^k(x_1 y_0 + x_0 y_1) \tag{20}$$

As in the previous case, we build a recursive algorithm as shown in fig. 3. Iterations that use formula (20) are pink while the Karatsuba iterations (18) are white. Assuming that $x_1 \cdot y_0$ and $x_0 \cdot y_1$ are also computed approximately, we get that the error of the approximation (20) is $\Delta(x \cdot y) = x \cdot y - \widehat{x \cdot y} = x_0 y_0 + 2^k(\Delta(x_1 \cdot y_0) + \Delta(x_0 \cdot y_1)) \in [0; 2^{2k} + 2^k(\Delta(x_0 \cdot y_1) + \Delta(x_1 \cdot y_0)))$. Using this, we can bound the error $\Delta(x \cdot y)$ of the recursive algorithm in fig. 3 as follows

$$\begin{aligned} 0 \leq \Delta(x \cdot y) < 2^{2k_0} + 2^{k_0}(\Delta(z_1) + \Delta(z_2)) < \\ 2^{2k_0} + 2^{k_0}(2^{2k_{12}} + 2^{2k_{13}} + 2^{k_{12}}(\Delta(z_{22}) + \Delta(z_{23})) + \\ 2^{k_{13}}(\Delta(z_{32}) + \Delta(z_{33}))) < \cdots = \Delta^{\{k_{ij}\}} \end{aligned} \tag{21}$$

Here $\Delta^{\{k_{ij}\}}$ is the upper bound on the error $\Delta(x \cdot y)$ which depends on every k_{ij}. This means that when we use the algorithm in fig. 3 in formula (17), $\hat{l_1}$ will be computed with an additional error of $\left\lceil \frac{\Delta^{\{k_{ij}\}}}{2^n} \right\rceil$. Thus the total error is upper bounded by

$$e(\hat{l_1}) < 3 + \left\lceil \frac{\Delta^{\{k_{ij}\}}}{2^n} \right\rceil \tag{22}$$

C. Optimizing FPGA resources for integer multipliers

Two out of three multipliers in figure 1 are multiplications by constants (m and q). This fact can be used to perform extra optimizations at the hardware level. Since our target hardware for this work is FPGA, we now present FPGA specific analysis. The technique can be replicated for other hardware with a different set of constraints.

The main basic logic blocks of FPGAs are DSP blocks and Look-Up Tables (LUTs). A straightforward way to implement integer multiplication at the leaves of the Karatsuba tree is by using DSP blocks. This is the most efficient way to multiply two unknown numbers. However, because the constants are known in advance, the cost can be reduced by implementing some of the multiplications using LUTs instead.

The exact resource optimization depends on the capacities of DSP and LUT of a specific FPGA. The general cost function for multiplication by a constant A can be defined recursively as:

$$C_t(A, \{k_{ij}\}) = \begin{cases} 1 & LUTs(A) > t \text{ and } bits(A) \leq DSP\_size, \\ LUTs(A)/t & LUTs(A) \leq t, \\ C_t(A_0, \{k_{ij}\}) + C_t(A_1, \{k_{ij}\}) + C_t(A_2, \{k_{ij}\}) & \text{else} \end{cases} \tag{23}$$

Here A_0, A_1, A_2 are the three Karatsuba terms which depend on k_{ij} and on the Karatsuba equation that is used (either (18), (19) or (20)). $LUTs(A)$ is a function that returns the number of LUTs required to implement the multiplication by a constant A. $bits(A)$ returns the bit-length of A. DSP_size is the maximum bit-length of the input to a DSP block. This recursive cost function is linear in the number of LUTs and has a hard threshold t that can be interpreted as the number of LUTs, above which it's better to use a DSP block instead.

The Karatsuba tree enables some flexibility in the choice of $\{k_{ij}\}$. Once we have this search space and a cost to be minimized, we can define and solve the optimization problems for q (the LSB multiplier on fig. 2) and m (the MSB multiplier on fig. 3):

$$\begin{aligned} &\underset{\{k_{ij}\}}{\text{minimize}} && C_t(q, \{k_{ij}\}) \\ &\text{subject to} && n + \sigma_l \leq 2k_i + k_{i-1} + k_{i-2} + \cdots + k_0 \end{aligned}$$

$$\begin{aligned} &\underset{\{k_{ij}\}}{\text{minimize}} && C_t(m, \{k_{ij}\}) \\ &\text{subject to} && n - \sigma_m \geq 2k_i + k_{i-1} + k_{i-2} + \cdots + k_0 \end{aligned}$$

The problems are solved by iterating on the splitting options $\{k_{ij}\}$ using dynamic programming. In the end, each product with cost 1 is implemented as a DSP block and if the cost is lower, LUTs are used. The threshold t is chosen such that the total number of DSP blocks is equal to the available amount. σ_m is chosen such that the error bound $\Delta^{\{k_{ij}\}}$ in (21) stays small enough. σ_l is found using (9) after the resulting error bound is calculated from (22). See appendix A for further details concerning LUTs and a demonstration of the technique for a specific FPGA.

III. Elliptic curve addition: Complete formulae

Many elliptic curve addition formulae are not complete, meaning they break for certain special cases, like adding two equal points or adding a point at infinity. To avoid handling these exceptions, we work with complete formulae that were originally introduced by [24]. Despite them being available for a decade or so now, they are not employed in practice since they have a larger number of modular operations as compared to other EC addition formulae in different coordinate systems [25].

Unfortunately, many of the known EC addition formulae in Jacobian or affine coordinates that are used in practice lead to high latency. In this paper, we show that the complete formulae [24] in homogeneous projective coordinates lead to a low-latency implementation in hardware. We follow the presentation of the formulae discussed in [26] (algorithm 7) for prime order curves. Although our techniques can be applied to many different curves, in this paper we consider a Barreto-Lynn-Scott type pairing-friendly curve BLS12-377 [27]–[29]. The equation for $\mathbb{G}_1$ of BLS12-377 in Weierstrass form is given by

$$y^2 = x^3 + 1 \tag{24}$$

where (x, y) are the affine coordinates of a point. The base field $\mathbb{F}_q$ is given by a 377-bit prime q[1] and the scalar field $\mathbb{F}_p$ is given by a 253-bit prime p. Exact values of p and q can be found in [27].

We rewrite (24) in homogeneous projective coordinates by substituting $(x, y) \to (X/Z, Y/Z)$ and the curve takes the form

$$Y^2 Z = X^3 + Z^3 \tag{25}$$

[1]meaning that every number and every equation in this section is modulo q

The complete formulae for adding points $P : (X_1, Y_1, Z_1)$ and $Q : (X_2, Y_2, Z_2)$ are given by

$$\begin{aligned} X_3 &= (X_1Y_2 + X_2Y_1)(Y_1Y_2 - 3Z_1Z_2) \\ &\quad - 3(Y_1Z_2 + Y_2Z_1)(X_1Z_2 + X_2Z_1) \\ Y_3 &= (Y_1Y_2 + 3Z_1Z_2)(Y_1Y_2 - 3Z_1Z_2) \\ &\quad + 9X_1X_2(X_1Z_2 + X_2Z_1) \\ Z_3 &= (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + 3Z_1Z_2) \\ &\quad + 3X_1X_2(X_1Y_2 + X_2Y_1) \end{aligned} \tag{26}$$

Naively, formulae (26) appear to require 15 modular multiplications, 13 additions, and 4 multiplications by 3. Figure 4 shows how the same computation can be done using just 12 multiplications, 17 additions, and 3 multiplications by 3. Due to its repetitive nature and homogeneity in the degree of coordinates, these formulae are highly parallelizable and hardware-friendly.

The circuit in fig. 4 is implemented in a pipeline and optimized for low latency. It takes in as inputs points P and Q and outputs coordinates (X_3, Y_3, Z_3) of the result $P + Q$.

IV. Multi-Scalar Multiplication (MSM)

This section is independent of the specific curve or field in question. Henceforth, we will provide a general description. Let $\mathcal{G}$ be an elliptic curve group of prime order p. Let $G = [G_0, G_1, \ldots, G_{N-1}] \in \mathcal{G}^N$ and $x = [x_0, x_1, \ldots, x_{N-1}] \in \mathbb{F}_p^N$ be N-element vectors of elliptic curve points and scalars, respectively. As mentioned in the introduction, MSM is a problem of computing

$$\begin{aligned} MSM(x, G) = \sum_{n=0}^{N-1} x_n \cdot G_n = \\ x_0 \cdot G_0 + x_1 \cdot G_1 + \cdots + c_{N-1} \cdot G_{N-1} \end{aligned} \tag{27}$$

Note that plus signs here refer to the elliptic curve addition. The scalar multiplication $x_n \cdot G_n$ refers to adding G_n to itself x_n times. Denote by

$$b = \lceil \log_2 p \rceil \tag{28}$$

the maximum number of bits in x.

As we discussed in the introduction §I, state-of-the-art algorithm to solve the MSM problem (27) is known as the Pippenger algorithm [1], and its variant that is widely used in the ZK space is called the bucket method [30] (see "Overlap in the Pippenger approach" in section 4). We describe it and our proposed improvements in the following section. There, we treat EC additions (PADD) and doublings (PDBL) as atomic operations, and considering our usage of complete formulae (26), we assume the costs of PADD and PDBL to be equal.

A. The bucket method

Let us partition each x_n from equation (27) into K parts such that each partition consists of c bits and $K = \lceil \frac{b}{c} \rceil$.

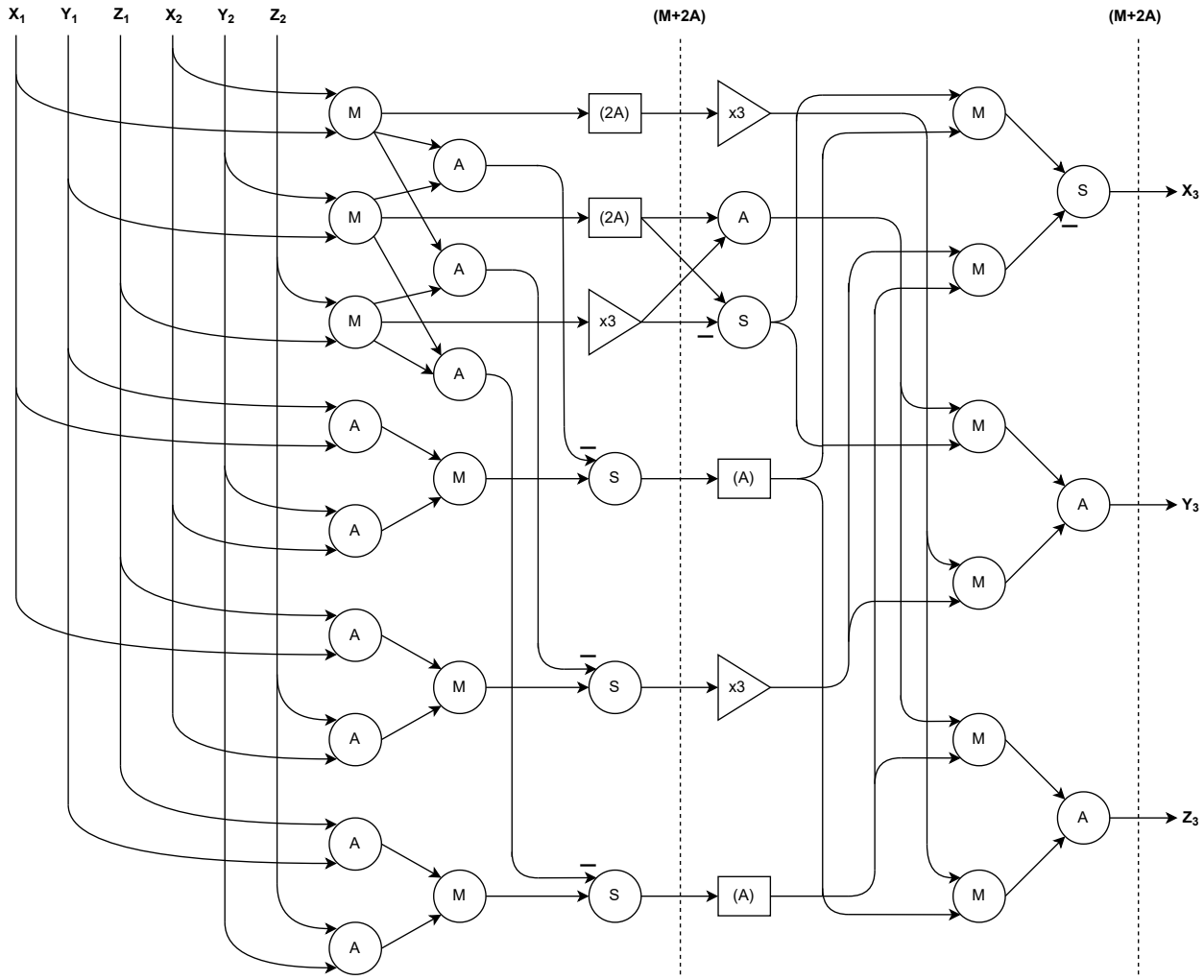

Fig. 4. Low latency elliptic curve addition module. $M, A/S$ refer to Multiplier and Adder/Subtractor, respectively. Triangles are multipliers by 3 and rectangles show the delays that occur due to sampling of intermediate values.

Denoting by $x_n^{[k]}$ the kth partition of x_n, equation (27) can be rewritten as

$$G = \sum_{n=0}^{N-1}\sum_{k=0}^{K-1} 2^{kc} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} \sum_{n=0}^{N-1} x_n^{[k]} G_n = \sum_{k=0}^{K-1} 2^{kc} G^{[k]} \tag{29}$$

where $G^{[k]} = \sum_{n=0}^{N-1} x_n^{[k]} G_n$ is the result of computing multi-scalar-multiplication with c-bit scalars $\{x_n^{[k]}\}$. Clearly, if we computed $G^{[k]}$ for each k, then the last step in (29) can be completed using *Horner*'s rule as

$$G = 2^c(...(2^c(2^c G^{[K-1]} + G^{[K-2]}) + G^{[K-3]})...) + G^{[0]} \tag{30}$$

Thus we reduced the MSM problem to computing the sums $G^{[k]}$. The total number of operations required in (30) can be broken down as follows:

1) Computing K partial sum $G^{[k]}$
2) $K-1$ PADDs attributed to the summations
3) $c(K-1)$ PDBLs attributed to the 2^c multiplications

We now proceed to describe an efficient algorithm to calculate $G^{[k]}$. The algorithm involves iterating over G_n, placing each one in a "bucket" corresponding to its coefficient $x_n^{[k]}$. Clearly, the number of non-zero buckets is $2^c - 1$. Per step, each bucket $l \in [1, 2^c - 1]$ acts as an accumulator, adding an assigned G_n to its current sum $B_l^{[k]}$. Buckets initiate at zero. For each k, the number of PADDs to compute all $B_l^{[k]}$ is $N + 1 - 2^c$. The partial sum $G^{[k]}$ is calculated as

$$G^{[k]} = \sum_{n=0}^{N-1} x_n^{[k]} \cdot G_n = \sum_{l=1}^{2^c-1} l \cdot B_l^{[k]} \tag{31}$$

which can be computed efficiently as the sum of the following recursive series

$$S_l^{[k]} = \sum_{i=1}^{l} B_i^{[k]} = S_{l-1}^{[k]} + B_l^{[k]} \tag{32}$$

thus

$$G^{[k]} = \sum_{l=1}^{2^c-1} S_l^{[k]} \tag{33}$$

We present the bucket method in algorithm 1. For the convenience of presentation, we partition the bucket method algorithm into three distinct parts, referred to as Loop 1, 2, and 3 hereafter.

Algorithm 1 Basic bucket method

Set $B_l^{[k]} = 0,\ \forall k \in [0, K-1],\ \forall l \in [1, 2^c - 1]$
▷ **The 1st loop**
for $n = 0, 1, \ldots, N-1$ **do** ▷ N inputs
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    Set $l = x_n^{[k]}$ ▷ Continue (skip) if $l = 0$
    $B_l^{[k]} \leftarrow B_l^{[k]} + G_n$ ▷ Partial bucket sums $B_l^{[k]}$
  end for
end for

Set $S^{[k]} = 0, G^{[k]} = 0$ ▷ **The 2nd loop**
for $l = 2^c - 1, 2^c - 2, \ldots, 1$ **do** ▷ Loop over buckets
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    $S^{[k]} \leftarrow S^{[k]} + B_l^{[k]}$
    $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$ ▷ Partial sums $G^{[k]}$
  end for
end for

Set $G = 0$ ▷ **The 3rd loop**
for $k = K-1, K-2, \ldots, 0$ **do** ▷ K partial sums
  $g \leftarrow 2^c G + G^{[k]}$ ▷ Horner's rule
end for

Cost of algorithm 1: One way to define the cost of 1 is to count the number of required EC operations. In figure 5 we plot this number against the parameter c and show the optimal values of c for various input sizes N.

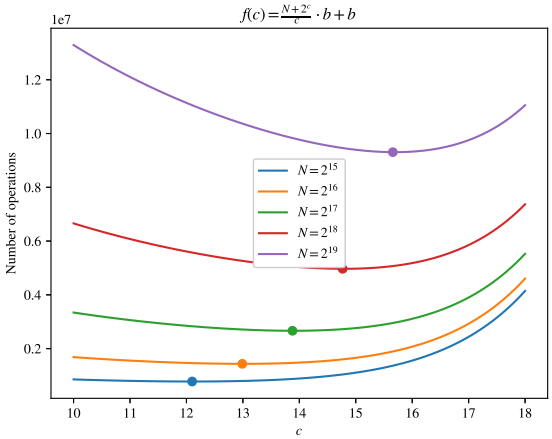

Fig. 5. The number of EC operations vs c, assuming $b = 253$ bits for various input sizes N. Dots represent the values of c that minimize $f(c)$.

A different way to define the cost of an algorithm is the expected span in clock cycles it requires to run once from start to completion. We'll use this cost function throughout the text. To that end, consider a fully pipelined PADD/PDBL kernel with a total delay of d clock cycles. This means that the kernel handles inputs for one EC operation every clock cycle and the result is returned after d clock cycles.

Loop 1 runs $(N - 2^c)K$ times with the delay of 1 between two consecutive iterations. The cost of Loop 1 is thus $(N - 2^c)K = \frac{(N-2^c)b}{c}$. In Loop 2, we need to perform $2^{c+1}K$ EC operations, each taking d clock cycles and $2K$ of them can be performed in parallel at a time (assuming $2K$ is less than d which is true in practice). The cost of Loop 2 is thus $\frac{2^{c+1}Kd}{2K} = d2^c$. Loop 3 contains $(K-1)(c+1)$ serial EC operations, taking around db clock cycles. Summing the costs of all loops provides the following cost function

$$f(c) = (N - 2^c)bc^{-1} + d(2^c + b) \tag{34}$$

B. An improvement to Loop 2 - segmentation of buckets

While the first loop in algorithm 1 can be parallelized, the second loop consists of K highly sequential computations, and only $2K$ parallel additions can be done at a time. To solve this issue, we break the second loop into M segments. Thus, instead of computing (31) we compute a segmented version of the buckets

$$\begin{aligned} G^{[k]} = &\left(1 \cdot B_1^{[k]} + 2 \cdot B_2^{[k]} + \cdots + U \cdot B_U^{[k]}\right) \\ &+ \left((U+1) \cdot B_{U+1}^{[k]} + (U+2) \cdot B_{U+2}^{[k]} + \cdots + 2U \cdot B_{2U}^{[k]}\right) \\ &+ \ldots \\ &+ \Big((U(M-1)+1) \cdot B_{U(M-1)+1}^{[k]} + \\ &(U(M-1)+2) \cdot B_{U(M-1)+2}^{[k]} + \cdots + (2^c - 1) \cdot B_{2^c-1}^{[k]}\Big) \end{aligned} \tag{35}$$

where $U = \lceil \frac{2^c-1}{M} \rceil$. We compute each of M segment sums separately, which can be done in parallel using the algorithm 2. Then the results are aggregated to obtain $G^{[k]}$.

Trade-offs: In the basic version 1, the computation of each $G^{[k]}$ includes $2 \cdot (2^c - 2)$ PADD. At every moment in time, 2 of these can be performed in parallel. In the segmented version, $2 \cdot (2^c - 2 - M)$ PADD are performed; $2M$ can be performed in parallel, and we suppose that M is large enough to fully occupy the pipelined adder. $\log_2(U)$ more PDBL[2] and $3M$ PADD need to be performed to get the final result $G^{[k]}$. Thus, the segmented version requires $K(M + \log_2(U))$ more EC operations while making most of the second loop more parallelizable. We get the total cost in clock cycles as

$$f(c) = (N + 2^c)bc^{-1} + d\,(2M + K + b) + 2K(2^c - 2 - M) \tag{36}$$

C. Simulating the bucket method

In this section, we discuss the performance of the segmented bucket method presented in algorithm 2. As mentioned above,

[2]Here we assume that U is a power of two so that computing the scalar multiplication by U only takes $\log_2(U)$ PDBL.

Algorithm 2 Buckets segmentation for the 2nd loop

▷ **The 2nd loop**
Set $S^{[k,m]} = 0, G^{[k,m]} = 0$ ▷ **2.0**
Def $U \equiv \frac{2^c}{M}$ ▷ Assume that M is a power of two
for $u = 1, 2 \ldots, U$ **do** ▷ U segment buckets
  for $m = 0, 1, \ldots, M-1$ **do** ▷ M segments
    Set $l = U(M-m) - u$ ▷ Skip if $l = 0$
    for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
      $G^{[k,m]} \leftarrow G^{[k,m]} + S^{[k,m]}$
      $S^{[k,m]} \leftarrow S^{[k,m]} + B_l^{[k]}$
    end for
  end for
end for

Set $S^{[k]} = 0$ ▷ **2.1**
Def $S^{[k,-1]} \equiv 0$
for $m = 0, 1, \ldots, M-2$ **do** ▷ $M-1$ segment
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    $S^{[k,m]} \leftarrow S^{[k,m]} + S^{[k,m-1]}$
    $S^{[k]} \leftarrow S^{[k]} + S^{[k,m]}$
  end for
end for

Def $v \equiv \log_2 U$ ▷ **2.2**
for v **cycles do** ▷ M segments
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    $S^{[k]} \leftarrow S^{[k]} + S^{[k]}$
  end for
end for

Set $G^{[k]} = 0$ ▷ **2.3**
for $m = 0, 1, \ldots, M-1$ **do** ▷ M segments
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    $G^{[k]} \leftarrow G^{[k]} + G^{[k,m]}$
  end for
end for

▷ **2.4**
for $k = 0, 1, \ldots, K-1$ **do** ▷ Loop over K partial sums
  $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$
end for

we work with the BLS12-377 curve [27], [28]. So the scalars $x_i \in \mathbb{F}_p$ in (27) are 253 bit integers, while the base field elements $G_i \in \mathbb{F}_q$ are 377 bit integers.

We assume that both scalars and points are variable inputs. In fig. 6, we present a high level design that executes the algorithm 2 in three phases

1) Bucket accumulation
2) Segments sum
3) Final accumulation

For the simulations in this section, we choose the number of segments in Loop 2 to be $M = 8$. The parameter c is varied. In our example in fig. 6, $c = 9$ so the design includes 29 bucket accumulators that are divided into 8 segments, each of them enclosing 64 buckets.

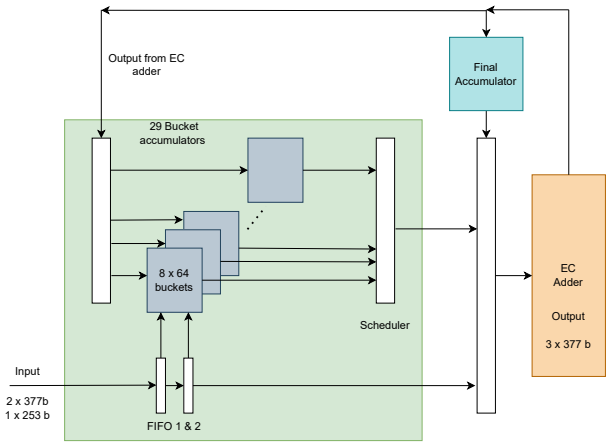

Fig. 6. Bucket method module for $c = 9$.

The EC adder module is a hardware unit that executes the addition of projective coordinates as described in §III. For the purposes of this section the EC adder module is a black box that works for d clock cycles before returning the result. However, the EC adder can receive a new task every clock.

Once the input enters the bucket accumulator module (the green box in fig. 6), the point is distributed to the buckets, defined by its scalar. If the bucket is empty then the input is written to it. If the bucket already contains a point then the accumulator sends an addition task to the EC adder with the current value and the new input. The result is sent back to the appropriate bucket. The bucket accumulator handles the output from the EC adder in higher priority than new input data. In Loop 2, the points in each of the segments are scheduled to be summed by the EC adder module. Thus, the output of the EC adder is sent back to the segments until all the data in the segments is exhausted, following which the data is sent to the final accumulator at the end of Loop 2. The final accumulator accumulates the results of all the segmented sums and uses the EC adder to perform Loop 3. While the final accumulator functions, the bucket accumulators are free to handle the next task.

Our operating design for the EC adder discussed in §III gives a delay of $d = 115$ clocks. Using a simulator written in Python and our design for input sizes $N = 2^{15}, 2^{17}$ and find that the optimum run time is obtained for $c = 12, 14$ respectively in tight agreement with our theoretical prediction plotted in fig 5. Our simulated results for $N = 2^{15}$ and $N = 2^{17}$ are plotted in fig 7.

Furthermore, it is possible to squeeze in more than one EC adder in our design. To get a sense of how efficient that would be, we plot the rate of MSM computation as a function of c in fig. 8. For optimal c, we see that the time for computing an MSM almost halves with the introduction of the second EC adder. Our simulations also indicate that the idle rate for two EC adders stays insignificant (see fig 9) and can be neglected.

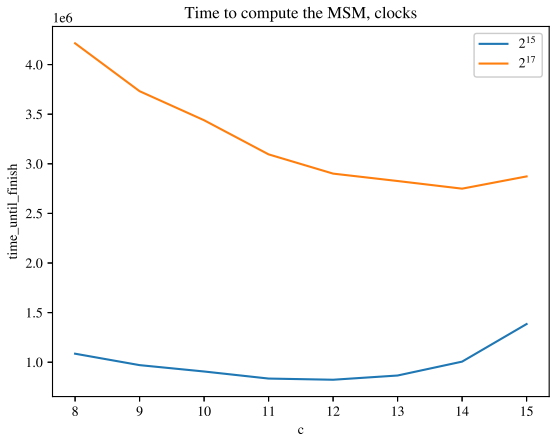

Fig. 7. We have plotted the bucket module run to finish vs c for a input vector of size $N = 2^{15}$, $N = 2^{17}$ and we find that the optimal run time is obtained when $c = 12$ and $c = 14$ respectively.

V. Implementation

In our current implementation, we use the design depicted in fig. 6. We choose parameters $c = 12, M = 8$, where M is the number of segments from algorithm 2. We use a single EC adder module, its delay is equal to 115 clock cycles. The clock speed is set to 125MHz.

To compare our implementation with the state-of-the-art GPU implementation, Sppark [18], we measure the delay and peak power draw of both. For our implementation we used Xilinx Alveo U55C, and for Sppark we used the Nvidia RTX3090 family, chosen as they outperformed all other GPU cards we tested: A100, V100, RTX5000, RTX6000.

Our benchmark on the two systems is done by taking two equally-sized vectors of scalars and EC points as inputs. It is worth noting that one likely scenario is for the EC points vectors to be precomputed in memory which reduces significantly the total running time[3]. Additionally we compute energy consumption per MSM which measures the monetary costs of computation. The performance is measured on MSMs of sizes 2^{15} to 2^{20}. The results, which agree with our simulations (see subsection IV-C), are summarized in table III.

As can be seen, our implementation is significantly better in terms of the peak power consumption and competitive in terms of delay, especially for smaller sizes. Also note that the computation time for Sppark grows slower than one might expect. This is due to the under-utilization of resources by the GPU for smaller-sized MSMs, as evident from the growing power consumption.

For future work we suggest three directions for improving our implementation:

1) *Increase clock frequency*: For our proof of concept we used clock frequency of 125MHz. With some engineering effort we believe it can be increased up to 500MHz. This immediately results in 4x speed improvement.
2) *Add EC adders*: As seen on figures 8 and 9, adding one more EC adder, doubles the performance.
3) *Further algorithmic performance improvements*: See section VI.

[3]Measuring running time at the host includes time to write to the hardware. Since the hardware part runs fast, we are sensitive to how the host-hardware interface is implemented and how much data goes through.

VI. Improvement proposals

In this section, we present potential improvements we have not yet implemented in hardware.

A. Signed scalars

Here we describe a method that utilizes the cheap negation in elliptic curve groups. In affine coordinates $P : (x, y)$ we can write

$$P \to -P \equiv (x, y) \to (x, -y) \tag{37}$$

Starting from the definition of the basic bucket method (29) we note that $x_n^{[k]} \in [1, 2^c - 1]$. To make use of the property (37), we need negative scalars, so we aim to shift the scalar set: $x_n^{[k]} \in [-2^{c-1}, 2^{c-1} - 1]$. A simple way to do this is to parse through $x_n^{[0]}, x_n^{[1]}, \ldots, x_n^{[K-1]}$ and, if $x_n^{[k]}$ is larger than $2^{c-1} - 1$, subtract 2^c and increment the $k+1$'st scalar by 1.

If the highest $K-1$'st digit is greater than 2^{c-1} then there is a potential overflow. In the case of BLS12-377 [27] 253-bit scalars will have some extra bits to offset the overflow provided c does not divide 253 (so, $c \neq 11$ and $c \neq 23$). The algorithm for shifting between the representations is given in algorithm 3 and can be run on the fly in Loop 1 of algorithm 1.

Algorithm 3 Conversion of c-bit scalars to signed scalars

input: $x_n^{[k]} \in [0, 2^c - 1]\ \forall k \in 0, 1, \ldots K-1$
for $k = 0, 1, \ldots K-1$ **do**
  if $x_n^{[k]} \geq 2^{c-1}$ **then**
    $\tilde{x}_n^{[k]} \leftarrow x_n^{[k]} - 2^c$ ▷ Subtract 2^c
    $\tilde{x}_n^{[k+1]} \leftarrow 1 + x_n^{[k+1]}$ ▷ Add 1 to the next scalar
  end if
end for
return $\tilde{x}_n^{[k]} \in [-2^{c-1}, 2^{c-1} - 1]\ \forall k \in 0, 1, \ldots K-1$

In Loop 1, we include a check for the sign of the scalar, which leads to a decrease in the number of buckets: from 2^c to 2^{c-1}. The rest of the algorithm can proceed as before including the segmentation improvement 2. The general idea is illustrated in algorithm 4. The new cost is

$$f(c) = \left(N + 2^{c-1}\right)\frac{b}{c} + d\left(\log_2\left(\left\lceil\frac{2^{c-1}-1}{M}\right\rceil\right) + 2M + K\right) \tag{38}$$

This allows a free (memory-wise) increase $c \to c+1$, decreasing the cost of the Loop 1 by a factor of $\frac{c+1}{c}$.

Finally, we mention that negation is just one example of the so-called "endomorphisms" which can be cheaply computed on curves of the BLS family. For an overview of other such endomorphisms and difficulties in applying them to the bucket method, see [31].

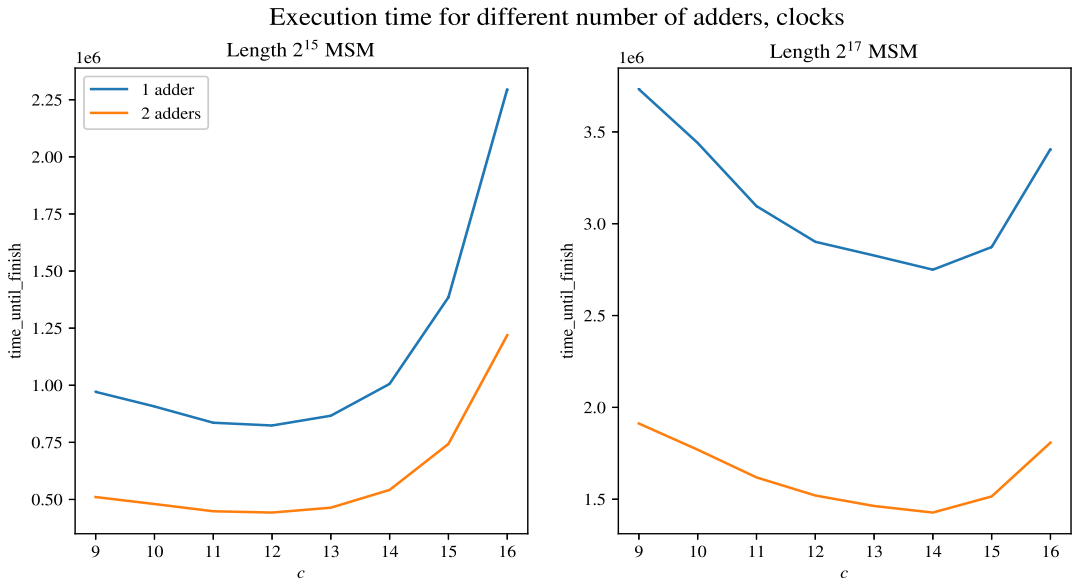

Fig. 8. Execution times for the sizes $N = 2^{15}$ and $N = 2^{17}$ respectively for various c. We note that the rate almost doubles when using 2 EC adders.

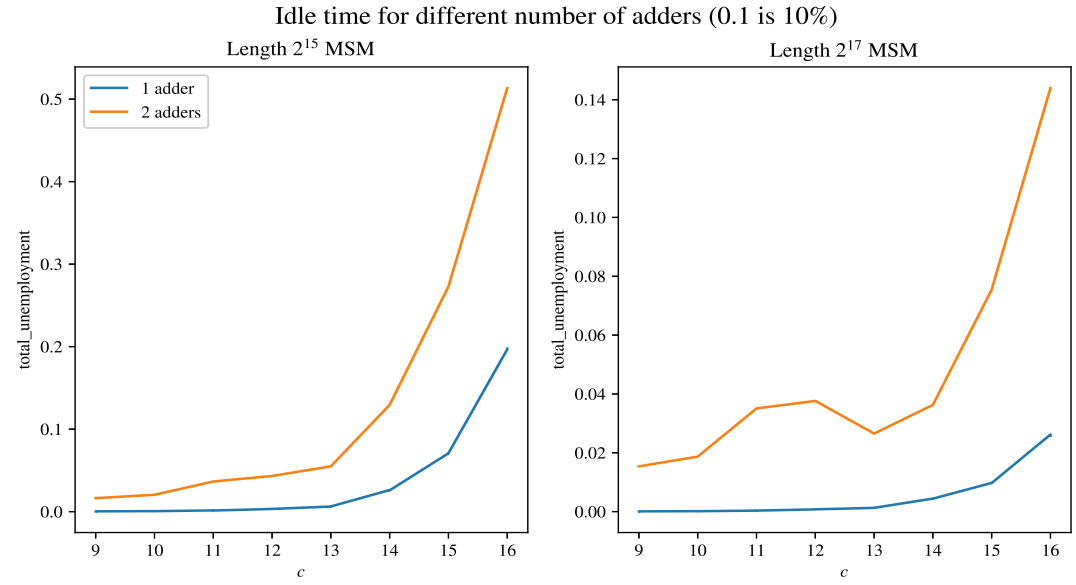

Fig. 9. Idle rates of EC adders for $N = 2^{15}$ and $N = 2^{17}$ respectively for various c. We note that the idle rate increases with the increase in the number of EC adders

B. Batched bucket method

This is a proposed improvement to the first loop of algorithm 1, which also requires a change in the way we do EC addition. Namely, we switch to representing points in affine coordinates. Since the affine addition includes field inversion, we first discuss how to offset the high computational costs of inversions using the so-called Montgomery trick.

1) Montgomery trick: Suppose that we want to invert $m > 1$ numbers: $a_1, a_2, \ldots, a_m$ modulo q. We can avoid performing many computationally costly inversions by using the so-called Montgomery trick. It is summarized in algorithm 5. It is easy to see that algorithm 5 performs $3(m-1)$ multiplications and 1 inversion instead of m inversions. Since in most cases modular multiplication is more than 3 times faster than inversion, the Montgomery trick is a very useful maneuver, provided that several field elements can be inverted "in parallel". For instance, it does not work if the input to one inversion depends on the other inversion being performed first.

The classical Montgomery trick as defined above is serial. It is, however, possible to parallelize the Montgomery trick, as shown in algorithm 6.

Analyzing this algorithm, we see that the first loop requires $L(R-1)$ multiplications. The inner Montgomery trick used to invert the final partial product in each batch takes $3(L-1)$ multiplications. Finally, the second loop takes $2L(R-1)$

Implementation	Hardware	Length of MSM	Time, ms.	Peak power, watts	Energy per MSM, joules
This work	U55C	2^{15}	9.8	18.4	0.174
		2^{16}	17.6	29.1	0.414
		2^{17}	35.9	33.6	0.976
		2^{18}	68.8	34.8	2.15
		2^{19}	136.6	34.8	4.50
		2^{20}	273.0	34.9	9.23
Sppark [18]	RTX 3090	2^{15}	9.1	173	1.44
		2^{16}	10.6	210	2.15
		2^{17}	14.6	287	3.34
		2^{18}	21.4	282	5.39
		2^{19}	33.7	384	8.99
		2^{20}	53.9	465	14.5

TABLE III
THE COMPARISON OF OUR FPGA IMPLEMENTATION AND SPPARK

Algorithm 4 The bucket method with signed scalars

Set $B_m^{[k]} = 0,\ \forall k \in [0, K-1],\ \forall m \in [1, 2^{c-1}]$
▷ **The 1st loop**
for $n = 0, 1, \ldots, N-1$ **do** ▷ N inputs
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    Set $l = x_n^{[k]}$ ▷ Convert to signed scalars
    if $l \geq 0$ **then**
      $B_l^{[k]} \leftarrow B_l^{[k]} + G_n$ ▷ Partial bucket sums $B_l^{[k]}$
    else
      $B_{|l|}^{[k]} \leftarrow B_{|l|}^{[k]} - G_n$ ▷ Partial bucket sums $B_{|l|}^{[k]}$
    end if
  end for
end for

Set $S^{[k]} = 0, G^{[k]} = 0$ ▷ **The 2nd loop**
for $l = 2^{c-1}, 2^{c-1}-1, \ldots, 1$ **do** ▷ Loop over 2^{c-1} buckets
  for $k = 0, 1, \ldots, K-1$ **do** ▷ K partial sums
    $S^{[k]} \leftarrow S^{[k]} + B_l^{[k]}$
    $G^{[k]} \leftarrow G^{[k]} + S^{[k]}$ ▷ partial sums $G^{[k]}$
  end for
end for

Set $G = 0$ ▷ **The 3rd loop**
for $k = K-1, K-2, \ldots, 0$ **do** ▷ K partial sums
  $G \leftarrow 2^c G + G^{[k]}$ ▷ Horner's rule
end for

multiplications. In total, we need $3(LR-1) = 3(m-1)$ multiplications, which is the same number as for the regular Montgomery trick. Thus, we parallelized the Montgomery trick with no additional computational cost.

The cost of the parallelized version is additional memory. Note that the lists t_i, as well as data for the inner Montgomery trick, need to be stored. This amounts to storing $m + L$ field elements, L more than the non-parallel version (algorithm 5).

In general, one can recursively execute the parallel Montgomery trick with more than one layer, in a tree-like structure.

Algorithm 5 The Montgomery Trick

input: $a_i, \forall i = 1, 2, \ldots m$
$t \leftarrow [a_1,$ ▷ Compute partial products
  $a_1 \cdot a_2 \pmod q$,
  $a_1 \cdot a_2 \cdot a_3 \pmod q$,
  $\vdots$
  $a_1 \cdot a_2 \cdot a_3 \ldots a_{m-2} \pmod q$,
  $a_1 \cdot a_2 \cdot a_3 \ldots a_{m-2} \cdot a_{m-1} \pmod q]$
$A \leftarrow (t[m-2] \cdot a_m)^{-1} \pmod q$ ▷ List t is 0-based
for $j \in \{m, m-1, \ldots, 3, 2\}$ **do**
  $a_j^{-1} \leftarrow A \cdot t[j-2] \pmod q$
  $A \leftarrow A \cdot a_j \pmod q$
end for
$a_1^{-1} \leftarrow A$.

Algorithm 6 Parallelized Montgomery Trick

input: $a_i, \forall i = 1, 2, \ldots m$
for $i \in \{0, 1, \ldots, L-1\}$ **do**:
  $t_i \leftarrow [a_{iR+1},$ ▷ Parallel partial products in batches
    $a_{iR+1} \cdot a_{iR+2} \pmod q$,
    $\vdots$
    $a_{iR+1} \cdot a_{iR+2} \ldots a_{(i+1)R-2} \pmod q$,
    $a_{iR+1} \cdot a_{iR+2} \ldots a_{(i+1)R-2} \cdot a_{(i+1)R-1} \pmod q]$
  $A_i \leftarrow t_i[R-2] \cdot a_{(i+1)R} \pmod q$
end for

▷ Invert $\{A_i\}$ in-place using regular Montgomery trick
$A_0, A_1, \ldots A_{L-1} \leftarrow$ Montgomery$(A_0, A_1, \ldots A_{L-1})$
for $i \in \{0, 1, \ldots, L-1\}$ **do**:
  for $j \in \{R, R-1, \ldots, 3, 2\}$ **do**:
    $a_{iR+j}^{-1} \leftarrow A_i \cdot t_i[j-2] \pmod q$
    $A_i \leftarrow A_i \cdot a_{iR+j} \pmod q$
  end for
  $a_{iR+1}^{-1} \leftarrow A_i$
end for

In any case, the number of multiplications remains the same and the memory footprint in the extreme case of a binary tree will be $2m$, or 2 times larger than for the non-parallel version.

To use the Montgomery trick, we will abstract away the details of it's implementation. Instead, we'll think of it as a stack of field elements that, after being populated, can return the inverses of these elements in the reverse order. In pseudocode, we denote by .push(a) a method that pushes a into the stack; this corresponds to the loop at lines 2-9 of algorithm 6. Method .invert() computes the inner Montgomery trick at line 12. Finally, .iter() successively iterates over the inverses of inputs in reverse order and stops when the data structure is empty; this happens at lines 13-19 in 6.

2) Batched bucket method: We adopt the notation from the basic bucket method discussed in algorithm 1. The buckets $B_l^{[k]}$ now store a single elliptic curve point in affine coordinates. We assume that the buckets are also equipped with a method .pop() that returns the contents and empties the bucket. We also maintain a stack t of capacity t_{max} that will hold pairs of affine EC points and bucket indices. When the stack reaches its maximum capacity, the inner Montgomery trick is called. The stack is initialized empty and has standard methods like .len(), .push(), and .pop(). By M we denote the Montgomery trick stack discussed earlier.

Algorithm 7 Batched version of the Loop 1 of the bucket method

1: **for** $n = 0, 1, 2, \ldots, N-1$ **do** ▷ Receive n-th scalar
2: **for** $k = 0, 1, 2, \ldots K-1$ **do** ▷ Working with $G^{[k]}$
3: $l \leftarrow x_n \pmod{2^c}$ ▷ Bucket index
4:
5: **if** $l > 0$ **then** ▷ Ignore bucket 0
6: ▷ Batch invert only when stack t is full
7: **if** length$(t) = t_{max}$ **then**
8: M.invert() ▷ Inner Montgomery
9: **for** d **in** M.iter() **do**
10: $(A, B, \hat{k}, \hat{l}) \leftarrow t$.pop()
11: $B_{\hat{l}}^{[\hat{k}]} \leftarrow$ ECadd(A, B, d) ▷ Algorithm 8
12: **end for**
13: **end if**
14:
15: **if** $B_l^{[k]} =$ NULL **then** ▷ Empty bucket
16: $B_l^{[k]} \leftarrow G_n$ ▷ Write G_n into the bucket
17: **else** ▷ Non-empty bucket
18: $G \leftarrow B_l^{[k]}$.pop() ▷ Pop a point
19: M.push$(x_G - x_{G_n})$
20: t.push$((G, G_n, k, l))$
21: **end if**
22: **end if**
23:
24: $x_n \leftarrow \lfloor \frac{x_n}{2^c} \rfloor$ ▷ Removing c last bits from x_n
25: **end for**
26: **end for**

The EC addition referred to in line 11 of algorithm 7 is done using the standard affine formulae. The only difference is that the denominators are precomputed (see algorithm 8).

Algorithm 8 Affine addition of elliptic curve points

1: **input:** G_1, G_2, d
2: $(x_1, y_1) \leftarrow G_1$ ▷ Coordinates of the point G_1
3: $(x_2, y_2) \leftarrow G_2$ ▷ Coordinates of the point G_2
4: $\lambda \leftarrow (y_1 - y_2) \cdot d \pmod{q}$
5: $x_3 \leftarrow \lambda^2 - x_1 - x_2 \pmod{q}$
6: $y_3 \leftarrow \lambda(x_1 - x_3) - y_1 \pmod{q}$
7: **return** (x_3, y_3)

References

[1] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 1976, pp. 258–263.

[2] J. Thaler, "Proofs arguments and zero knowledge," https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf.

[3] T. Ingonyama, "Ingopedia," https://github.com/ingonyama-zk/ingopedia/blob/main/README.md.

[4] M. Petkus, "Why and how zk-snark works," 2019. [Online]. Available: https://arxiv.org/abs/1906.07221

[5] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," Cryptology ePrint Archive, Paper 2012/215, 2012, https://eprint.iacr.org/2012/215. [Online]. Available: https://eprint.iacr.org/2012/215

[6] V. Buterin, "Quadratic arithmetic programs from zero to hero," https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649.

[7] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.

[8] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit, "Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting," Cryptology ePrint Archive, Paper 2016/263, 2016, https://eprint.iacr.org/2016/263. [Online]. Available: https://eprint.iacr.org/2016/263

[9] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent snarks from dark compilers," Cryptology ePrint Archive, Paper 2019/1229, 2019, https://eprint.iacr.org/2019/1229. [Online]. Available: https://eprint.iacr.org/2019/1229

[10] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Paper 2019/953, 2019, https://eprint.iacr.org/2019/953. [Online]. Available: https://eprint.iacr.org/2019/953

[11] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," Cryptology ePrint Archive, Paper 2019/1047, 2019, https://eprint.iacr.org/2019/1047. [Online]. Available: https://eprint.iacr.org/2019/1047

[12] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," Cryptology ePrint Archive, Paper 2017/1066, 2017, https://eprint.iacr.org/2017/1066. [Online]. Available: https://eprint.iacr.org/2017/1066

[13] J. Groth, "On the size of pairing-based non-interactive arguments," Cryptology ePrint Archive, Paper 2016/260, 2016, https://eprint.iacr.org/2016/260. [Online]. Available: https://eprint.iacr.org/2016/260

[14] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez, "Lunar: a toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions," Cryptology ePrint Archive, Report 2020/1069, 2020, https://ia.cr/2020/1069.

[15] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323.

[16] P. L. Montgomery, "Modular multiplication without trial division," Mathematics of Computation 44, no. 170 (1985): 519–21., https://doi.org/10.2307/2007970.

[17] Z. Cao, R. Wei, and X. Lin, "A fast modular reduction method," Cryptology ePrint Archive, Paper 2014/040, 2014, https://eprint.iacr.org/2014/040. [Online]. Available: https://eprint.iacr.org/2014/040

[18] Supranational, "Sppark: Zero-knowledge template library," https://github.com/supranational/sppark.

[19] "Zcash fpga," https://github.com/ZcashFoundation/zcash-fpga.

[20] "Fpga snark prover targeting the bn128 curve," https://github.com/bsdevlin/fpga_snark_prover.

[21] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.

[22] "Fpga accelerator development and deployment in the cloud," https://aws.amazon.com/ec2/instance-types/f1.

[23] D. J. Bernstein, "Multidigit multiplication for mathematician," http://cr.yp.to/papers/m3.pdf.

[24] W. Bosma and H. Lenstra, "Complete systems of two addition laws for elliptic curves," *Journal of Number Theory*, vol. 53, no. 2, pp. 229–240, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022314X85710888

[25] D. J. Bernstein and T. Lange. [Online]. Available: http://www.hyperelliptic.org/EFD/

[26] J. Renes, C. Costello, and L. Batina, "Complete addition formulas for prime order elliptic curves," Cryptology ePrint Archive, Paper 2015/1060, 2015, https://eprint.iacr.org/2015/1060. [Online]. Available: https://eprint.iacr.org/2015/1060

[27] Arkworks, "Arkworks library for zksnarks," https://docs.rs/ark-bls12-377.

[28] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," Cryptology ePrint Archive, Paper 2018/962, 2018, https://eprint.iacr.org/2018/962. [Online]. Available: https://eprint.iacr.org/2018/962

[29] P. S. L. M. Barreto, B. Lynn, and M. Scott, "Constructing elliptic curves with prescribed embedding degrees," in *Security in Communication Networks*, S. Cimato, G. Persiano, and C. Galdi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 257–267.

[30] D. J. Bernstein, J. Doumen, T. Lange, and J.-J. Oosterwijk, "Faster batch forgery identification," Cryptology ePrint Archive, Report 2012/549, 2012, https://ia.cr/2012/549.

[31] G. Gutoski, zK hack : Youtube. [Online]. Available: https://www.youtube.com/watch?v=Bl5mQA7UL2I&ab_channel=ZeroKnowledge

APPENDIX
OPTIMIZATIONS IN MODULAR MULTIPLICATION FOR U55C

Here, we focus on optimizations for the hardware of our choice — Xilinx U55C FPGA.

Recall that FPGAs contain DSP blocks that can multiply 18-bit and 27-bit numbers. Using the Karatsuba tree, a 377-bit by 377-bit multiplication can be done with 162 DSP blocks.

As mentioned in section III, each EC adder is composed of 12 modular multipliers which are in turn composed of 3 377-bit by 377-bit integer multipliers. Implementing this naively, we will need 5832 DSP blocks for one EC adder or 11664 for two. However, Our FPGA barely have enough DSP blocks even for one EC adder. Therefore, replacing some of the DSP blocks with LUTs is necessary for building a working pipelined EC adder.

As an example of multiplying by constant, consider $A = 10001110000$, then $A \cdot B$ is:

$$A \cdot B = B \ll 10 + B \ll 6 + B \ll 5 + B \ll 4$$

This can be improved using the Canonical Signed Digit (CSD) representation. The constant A can be represented as a combination of a positive part and a negative part: $A = 10010000000 - 10000$.

Now the computation contains only 3 terms:

$$A \cdot B = B \ll 10 + B \ll 7 - B \ll 5$$

The average fraction of non-zero bits in the CSD representation is one-third as compared to one-half in the binary representation. The costs of addition and subtraction are the same LUT-wise and therefore we prefer the CSD representation.

All of the above leads to the conclusion that the DSP blocks that should be replaced are the ones performing multiplication by constants with low Hamming weight in the CSD representation. The basic LUT can implement any $5 \to 2$ logical function. Therefore addition/subtraction combinations of 2 or 3 bits are implemented with a single LUT. When we have 4 or 5 bits (this can happen only when the computation includes at least 4 or 5 terms), 2 LUTs are required. This logic can be extended to operations with larger bit sizes. This implies that the cost in LUTs of a multiplication depends not only on the Hamming weight of the constant but also on the non-zero bit distribution. Here are some examples. If we take a variable B with a bit-size of 10 bits, and different constants A:

1. $A = 10000000100$ translates into $B \ll 10 + B \ll 2$. The cost is 2 LUTs because there are 2 terms with an overlap of 2 . The LUTs are for computing $b_0 + b_8$ and $b_1 + b_9$ as seen in fig. 10 (The carry is taken care of by a Carry-8 module).
2. $A = 10001000000$ translates into $B \ll 10 + B \ll 6$. The cost is 6 LUTs because there are 2 terms with an overlap of 6 as shown in fig. 11.

$$\begin{array}{r} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\ +\, b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\ \hline \end{array}$$

Fig. 10. Addition with an overlap of 2.

$$\begin{array}{r} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\ +\, b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 \\ \hline \end{array}$$

Fig. 11. Addition with an overlap of 6.

The same Karatsuba tree that we use for 377-bit numbers can actually be used for numbers up to 416 bits. This gives us some flexibility in the choice of $\{k_{ij}\}$. After computing optimal values of $\{k_{ij}\}$ by solving the optimization problems that are defined in II-C, we obtain the results depicted in tables IV and V.

MSB multiplication by m	**# of DSP**	**# of LUT**
No optimizations	162	9.5K
No optimizations, DSP replaced by LUTs	35	19.5K
Optimized k_{ij}, no MSB optimization	35	16K
Both optimizations	35	13.5K

TABLE IV
THE RESULTS OF OPTIMIZATIONS IN FIG. 3 AND IN II-C

After implementing all optimizations, we have that $\Delta^{\{k_{ij}\}} \approx 1.994 \cdot 2^{377}$. Refining error bounds using specific

LSB multiplication by q	**# of DSP**	**# of LUT**
No optimizations	162	9.5K
No optimizations, DSP replaced by LUTs	35	19K
Optimized k_{ij}, no LSB optimization	35	15.5K
Both optimizations	35	10K

TABLE V
THE RESULTS OF OPTIMIZATIONS IN FIG. 2 AND IN II-C

values of m and q, the maximal error $e(\hat{l_1})$ from equation (22) can be shown to be 4. This means that the only modification that needs to be applied to the algorithm in fig. 1 — potentially subtracting up to $4q$ (and not $3q$) in the last step.