IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 1, pp. 275–313. DOI:10.46586/tches.v2025.i1.275-313

A High-performance NTT/MSM Accelerator for Zero-knowledge Proof Using Load-balanced Fully-pipelined Montgomery Multiplier

Xiangren Chen[1], Bohan Yang[1,*], Wenping Zhu[1], Hanning Wang[1], Qichao Tao[1], Shuying Yin[1], Min Zhu[2], Shaojun Wei[1] and Leibo Liu[1,*]

[1]Beijing National Research Center for Information Science and Technology (BNRist), School of Integrated Circuits, Tsinghua University, China.
[2]Wuxi Micro Innovation Integrated Circuit Design Co., Ltd.
chen-xr23@mails.tsinghua.edu.cn,{bohanyang,zhuwp,wanghn,qichaotao,yinshuying,wsj,liulb}@tsinghua.edu.cn;zhumin@mucse.com
[*] Corresponding Author.

Abstract. Zero-knowledge proof (ZKP) is an attractive cryptographic paradigm that allows a party to prove the correctness of a given statement without revealing any additional information. It offers both computation integrity and privacy, witnessing many celebrated deployments, such as computation outsourcing and cryptocurrencies. Recent general-purpose ZKP schemes, e.g., zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK), suffer from time-consuming proof generation, which is mainly bottlenecked by the large-scale number theoretic transformation (NTT) and multi-scalar point multiplication (MSM). To boost its wide application, great interest has been shown in expediting the proof generation on various platforms like GPU, FPGA and ASIC.

So far as we know, current works on the hardware designs for ZKP employ two separated data-paths for NTT and MSM, overlooking the potential of resource reusage. In this work, we particularly explore the feasibility and profit of implementing both NTT and MSM with a unified and high-performance hardware architecture. For the crucial operator design, we propose a dual-precision, load-balanced and fully-pipelined Montgomery multiplier (LBFP MM) by introducing the new mixed-radix technique and improving the prior quotient-decoupled strategy. Collectively, we also integrate orthogonal ideas to further enhance the performance of LBFP MM, including the customized constant multiplication, truncated LSB/MSB multiplication/addition and Karatsuba technique. On top of that, we present the unified, scalable and high-performance hardware architecture that conducts both NTT and MSM in a versatile pipelined execution mechanism, intensively sharing the common computation and memory resource. The proposed accelerator manages to overlap the on-chip memory computation with off-chip memory access, considerably reducing the overall cycle counts for NTT and MSM.

We showcase the implementation of modular multiplier and overall architecture on the BLS12-381 elliptic curve for zk-SNARK. Extensive experiments are carried out under TSMC 28nm synthesis and similar simulation set, which demonstrate impressive improvements: (1) the proposed LBFP MM obtains 1.8× speed-up and 1.3× less area cost versus the state-of-the-art design; (2) the unified accelerator achieves 12.1× and 5.8× acceleration for NTT and MSM while also consumes 4.3× lower overall on-chip area overhead, when compared to the most related and advanced work PipeZK.

Keywords: ZKP · NTT · Montgomery multiplication · MSM

Received: 2024-07-15 Accepted: 2024-09-15 Published: 2024-12-09

1 Introduction

Introduced in the 1980s by Goldwasser et al. [GMR85], the concept of zero-knowledge proof allows one party to prove to another party that a certain statement is true without revealing any extra information. More formally, the statement refers to $F(x, w) = true$ with F being the function to be verified, x being the public inputs and w being the secret inputs only known by the prover. Within the last decade, general-purpose and non-interactive zero-knowledge proof schemes have made the jump from theory to practice. For instance, an efficient privacy protocol known as zk-SNARK (zero-knowledge Succinct Non-interactive ARgument of Knowledge) [Gro16, COS20, GWC19] has recently drawn much attention from academia and industry. Particularly, the property of succinctness is embodied by the fact that the size of proof is constrained to hundreds of bytes (e.g., 192 bytes [Gro16]) and the verification time is almost constant, regardless of the sizes of function F and inputs x/w. Other notable ZKP schemes encompass hash-based zkSTARK (Zero-Knowledge Scalable Transparent Argument of Knowledge) [BBHR19] that avoids the necessity of trusted setup and could resist the attack of quantum computer, and code-based Plonky2 [Tea22] that realizes highly performant recursion over Goldilocks field, and so forth. Recently, ZKP has witnessed many celebrated deployments, including computation outsourcing [ZGK+17], electronic voting [PR18], cryptocurrencies [BFV19], and machine learning [XZL+23]. The Defense Advanced Research Projects Agency (DARPA) also launches the Securing Information for Encrypted Verification and Evaluation (SIEVE) program that generates ZKP for defense capability and vulnerability disclosure [dar19].

However, the remarkable power of ZKP comes at a huge overhead within proof generation, mainly resulting from elliptic curve related computation like MSM or polynomial computations. Since almost all modern ZKP systems are built on polynomial commitment schemes, NTT ends up being one of the major bottlenecks in the proof-generation process, especially for those based on error-correcting codes, such as STARK and Plonky2. In pairing-based SNARK schemes, NTT and MSM respectively takes up approximately 30% and 70% time of the overall proof-generation process on software platforms [LWY+23]. The modular multiplication of large data-width serves as the crucial building block for both MSM and NTT as well. Undoubtedly, accelerating NTT and MSM is of great significance to enabling real-world infrastructures and capturing more complex applications for ZKP-related systems.

Related works on NTT & MSM. There is a succession of excellent works on devising individual hardware sub-systems for ZKP-oriented NTT [WG23, HMR23] and MSM [RDQY24, ZHLH23], respectively. However, the feasibility and profit of implementing them on the shared architecture are never discussed up to now. Compared to NTT used in the post-quantum cryptography (PQC) and fully homomorphic encryption (FHE), significantly larger vector length and bit-width feature that of ZKP, reminiscent of the massive off-chip memory access. As a consequence, data movement between off-chip memory and on-chip scratchpad has to be taken into consideration, which in turn influences the design strategy of on-chip NTT core. There are many large-scale NTT architectures for FHE [SFK+21, KKK+22, YZF+23, MAK+23], which provide valuable inspirations for the realm of ZKP. In contrast, the performance bottleneck of MSM lies in the on-chip computation instead of the off-chip memory access like NTT. Besides, the data-flow and execution time of MSM depend on the concrete data distribution, which is deemed to be dynamic and unpredictable.

PipeZK [ZWZ+21] marks the first attempt at crafting NTT and MSM hardware designs for zk-SNARK. It adopts the four-step NTT version and introduces a single-path delay feedback (SDF) pipelined architecture to ameliorate memory access pressure. But this architecture depletes substantial First-In-First-Out (FIFOs) to orchestrate address intervals at every stage of NTT, which yields a mere 50% hardware utilization. As for the MSM kernel, PipeZK adopts the Pippenger algorithm and presents a dynamic work

dispatch mechanism to share the heavy point adder units. Zprize2022 [Pru22] launches two special tracks for hardware acceleration of NTT and MSM on FPGA. However, the evaluation criteria primarily emphasize speed enhancement while overlooking optimizations of area overhead. The top contenders embrace the common strategy of combining parallel on-chip NTT architecture with dual-tier memory architecture. Notably, the leading team, Supranational, achieves remarkable feats, computing 2^{24} 64-bit point in just around 2 milliseconds for NTT. But the selected 64-bit NTT modulus falls short of accommodating pairing-based zk-SNARK, which typically demands modulus widths exceeding 253-bit. Also, the first-place team at MSM track presents a split CPU-FPGA MSM engine, which utilizes the fully-pipelined, strongly unified mixed point adder based on the Scaled Twisted Edwards curve [RDQY24]. The adopted point addition formula reduces the number of modular multiplication, but it should be performed on condition that the resulting sum is added to the raw point with affine coordinate. Thus, adding two resulting points both with extended coordinates is not supported by the hardware part. The recent multi-bank based MSM architecture [LZD+24], coupled with optimized scheduling mechanism, presents sizable performance gain. But it takes no account of the practical data movement from DRAM to on-chip SRAM, which in turn influences how the overall latency is calculated. PipeMSM [Xav22] also presents a pipelined design for MSM on FPGA, which is built upon the complete point addition formula with modified Barret modular multipliers. CycloneMSM [ABC+22] develops a heuristic-based scheduler to manipulate the order of operations during bucket aggregation, decently handling the colliding points to increase throughput. Nevertheless, the underlying large-width modular multipliers in these works still leave some room for improvement, which plays a vital role in the overall performance.

Contributions. Given that two separated hardware data-paths for NTT and MSM consume tremendous computation and memory resource, it is feasible and profitable to devise a unified and high-performance hardware architecture for them. Building on this holistic insight, our main contributions are summarized as follows.

- We analyse the role of NTT and MSM within a typical workflow of zk-SNARK protocol, further demonstrating that the data dependency between NTT and MSM accounts for the design rationale behind unification. Then, we identify several design challenges that are specific for a unified and efficient architecture supporting both NTT and MSM. (Section 2)

- We discover the inherent issue within prevailing Montgomery algorithms, i.e., the severe load imbalance when directly implemented in a fully-pipelined manner. Crucially, a simple, modularized yet effective technique called as mixed-radix form is proposed to tackle this issue, considerably saving pipeline registers and improving the area efficiency. Furthermore, we improve the prior quotient-delayed technique by achieving the lower bound of iteration count and circumventing extra post-correction, modifying the critical path at the algorithm level. Based on these new techniques, we develop the mixed-radix LBFP MM that supports dual-precision modular multiplication for both NTT and MSM. Collectively, we also integrate other optimization techniques to enhance the performance of LBFP MM, including the customized constant multiplication, truncated LSB/MSB multiplication/addition and Karatsuba technique. (Section 3)

- For the first time, we present a unified and high-performance architecture that supports both NTT and MSM by making a series of optimizations. In particular, a versatile ping-pong scheduling for NTT and a dedicated load-aware arbitration mechanism for MSM are proposed to maximize the resource utilization. A novel configurable PE array and compact memory partition strategy are proposed to intensively reduce the entire area overhead. (Section 4)

- We evaluate the implementation of modular multiplier and overall architecture instantiated with the BLS12-381 curve. The experimental results are carried out under TSMC 28nm synthesis and similar simulation set, which achieve record-breaking improvements: (1) the LBFP MM obtains 1.8× speed-up and 1.3× less area cost versus the state-of-the-art; (2) the unified accelerator delivers 12.1× and 5.8× acceleration for NTT and MSM while also saves 4.3× area overhead, when compared to the state-of-the-art counterpart PipeZK. (Section 5)

2 Preliminaries

2.1 The Role of NTT & MSM within ZKP

Both NTT and MSM stand as the core algebraic operations within many ZKP protocols. In this section, we use one of the most mature and prominent ZKP schemes, zk-SNARK, as an example to illustrate its specific role in the protocol. Generally, this protocol consists of two parties: a prover and a verifier. The prover can convince the verifier that, "Given a function F and input $\vec{a}$, I know a secret witness $\vec{b}$ such that $F(\vec{a}, \vec{b}) = \text{true}$," by generating a zk-SNARK proof. Three notable properties are provided by zk-SNARK: (1) succinctness, i.e., small proof sizes (< 1KB) and fast verification (< 10ms) independent of the application complexity F, (2) non-interactive, i.e., one-time message passing from the prover to the verifier, and (3) zero knowledge, meaning that the proof reveals no information about the secret $\vec{b}$ beyond the statement itself.

The parameters for pairing-based zk-SNARK schemes are derived from pairing-friendly elliptic curves, such as BLS12-381 [WB19] and BLS12-377 [AV20]. For instance, the equation of the BLS12-381 elliptic curve is defined as $E(\mathbb{F}_q) : y^2 = x^3 + 4 \bmod q$, where q is a 381-bit modulus, signifying that the coordinates of point $P = (x, y)$ (or generator $G_1 = (x, y)$) should take values from finite field $\mathbb{F}_q$. The typical group operations on elliptic curves include point doubling (PDBL) $P + P$, point addition (PADD) $P_1 + P_2$, and scalar point multiplication $s \cdot G$, where the scalar s is a 255-bit integer drawn from another finite field $\mathbb{F}_m$, and G denotes point generator for the elliptic curve. Another twisted BLS12-381 curve is defined as: $E(\mathbb{F}_{q^2}) : y^2 = x^3 + 4 \cdot (i + 1)$, with point generator $G_2 = (x_0 + i \cdot x_1, y_0 + i \cdot y_1)$ residing on $\mathbb{F}_{q^2}$. In zk-SNARK schemes, the computational subject of NTT refers to polynomials comprised of coefficients over the finite field $\mathbb{F}_m$, while the MSM is performed between the scalar vector over $\mathbb{F}_m$ and point vector over $\mathbb{F}_q/\mathbb{F}_{q^2}$. As a non-interactive ZKP protocol, zk-SNARK is fundamentally structured into three stages, with an overarching workflow depicted in Figure 1.

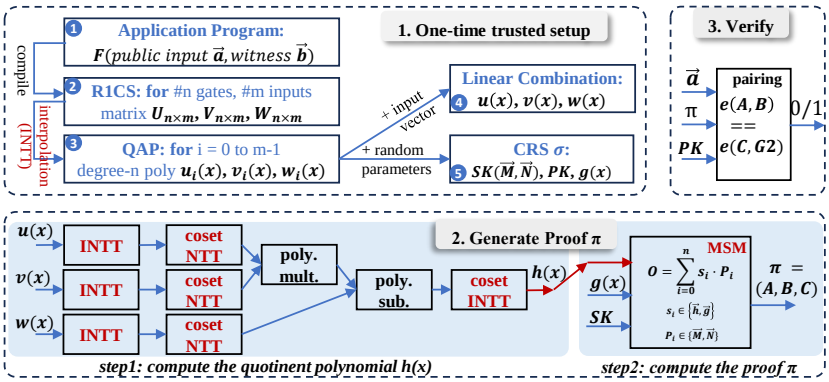


Figure 1: The typical work flow of zk-SNARK.

One-time trusted setup. For a given circuit program F with m inputs, a trusted

third party initially compiles it into a series of simple 2-input-1-output circuit gates, forming the so-called rank-1 constraint system (R1CS), which is subsequently transformed into three $n \times m$ binary matrices $U_{n\times m}, V_{n\times m}, W_{n\times m}$, with n on the order of 2^{20} in typical applications. Afterward, each column of the matrices is subject to interpolation transformations based on INTT, yielding the quadratic arithmetic program (QAP) in degree n-1 polynomial form: $u_i(x), v_i(x), w_i(x)$ for $i = 0, 1, ..., m-1$. Finally, QAPs are further normalized into three polynomials $u(x), v(x), w(x)$ by performing linear combination with input vectors: $u(x) = \sum_{i=0}^{m} a_i \cdot u_i(x)$, $v(x) = \sum_{i=0}^{m} a_i \cdot v_i(x)$, $w(x) = \sum_{i=0}^{m} a_i \cdot w_i(x)$. Also, a series of randomly chosen security parameters are used along with QAPs to generate the common reference string (CRS) σ for the entire ZKP system, which includes the polynomial $g(x)$, secret key (SK) and public key (PK) utilized by the prover and verifier.

Proving and verifying phases. The proving phase can be roughly divided into two steps, each primarily involving polynomial calculations and MSMs. The objective of the first step is to compute the quotient polynomial $h(x) = \frac{u(x)\cdot v(x) - w(x)}{x^n - 1}$. First, we need to utilize coset NTT to determine the point-value expressions of polynomials $u(x), v(x)$ and $w(x)$. Subsequently, operations including vector dot products and polynomial substractions are applied to the NTT-computed results to obtain the point-value expression of $h(x)$. Finally, the coset INTT is employed to obtain the coefficient vector of the quotient polynomial $h(x)$. In the second step of the proving phase, the MSM is conducted among the output of the first step $h(x)$, $g(x)$ and the private key SK (consists of multiple point vectors $\{\vec{M}, \vec{N}\}$), which generates a concise proof π with short size (e.g., 192-bit for BLS12-381 curve). Specifically, each MSM task can be expressed as: $O = \sum_{i=0}^{n} s_i \cdot P_i$, where the scalar vector $\vec{s}$ is taken from $\vec{h}$ or $\vec{g}$, and the point vector $\vec{P}$ refers to $\vec{M}$ or $\vec{N}$, respectively. The verifying phase takes in the public key PK, the proof π and the public inputs $\vec{a}$, leveraging the bilinear property of pairing to verify the correctness of proof. Typically, this computation process is comprised of Miller loop and final exponentiation, which can be completed within a few milliseconds.

2.2 Motivations & Design Challenges

Motivations. Several prior arts [ZWZ+21, Xav22] already reveal that designing separated data-paths for NTT and MSM incurs significant area overhead. From the perspective of data-flow dependency, NTT should precede MSM to compute the quotient polynomial $h(x)$, based on which MSM is conducted between the coefficient vector $\vec{h}$ and the precomputed point vector $\vec{P}$, as shown in Section 2.1. In other words, NTT and MSM over G_1 [LFG23] have to be executed serially, which results in MSM-related computation resources underutilized during NTT computation. One possible counter-example is the scenario that NTT could be performed for another proof when MSM is utilized to compute the current proof. However, concurrently performing two proof-generation processes would require two individual setup processes, which not only doubles the computation and memory requirement but also complicates the hardware controller design. In fact, most previous works [MXS+23, ZWZ+21] only compute one proof at a time, which can cater to many real-world application scenarios. On the other hand, even if both NTT and MSM could run in parallel, the overall speedup would still be minor. Because the computation latency occupied by NTT after hardware acceleration will be much smaller than that of MSM [ZWZ+21], making the overall performance primarily limited by the latency of MSM over G_1. Therefore, we seek to devise a unified and high-performance hardware architecture by reusing and configuring the computation and memory resources, solidly improving the area efficiency over state-of-the-art designs.

Challenges. However, implementing NTT and MSM with a unified architecture poses new design challenges. *For one thing, existing NTT and MSM computation paradigms have much difference in terms of data-flow and memory organization.* The data-flow of

NTT follows a fixed and iterative butterfly computation pattern, whose design challenges involve addressing potential bank access conflicts due to varying address strides. In contrast, the data-flow of MSM cannot be predicted in advance, which is related to the distribution of scalar vector. The core issue is the sophisticated controller design that manages the sources of inputs for the point adder unit to improve the hardware utilization. Besides, the pipelined SDF/MDC NTT architecture in prior works adopts segmented FIFOs with varying sizes, which are not conducive to a unified memory organization [ZWZ+21]. *For another thing, the bit widths of the underlying arithmetic units for NTT and MSM are typically quite different.* Taking the BLS12-381 curve as an example, the NTT is performed under 255-bit modulus from scalar field, whereas the coordinates of point vector in MSM utilize the 381-bit modulus. Hence, dual-precision arithmetic units are required to serve both MSM and NTT. Compared to modular addition and subtraction, designing a multi-precision modular multiplier is more challenging because we cannot simply pad the operands with zeros due to the impact of modular reduction. Although previous works employ the scalable Coarsely Integrated Operand Scanning (CIOS) Montgomery algorithm to implement multi-precision modular multipliers, this design approach introduces feedback loops into the pipelined data-path and thus is not fully pipelined. *It is underscored that the modular multiplier with fully-pipelined property can take in one data point per cycle to unlock the theoretically optimal throughput, otherwise the cycle count proportionally increases with the folded degree.* In summary, there still remains a conspicuous absence of area-efficient, fully-pipelined and dual-precision modular multiplier for large modulus.

2.3 Four-step NTT

Similar to FHE, ZKP serves as an application-layer protocol and thus its computational parameter closely correlates with the scale of the circuit program to be proved. As of typical applications, the vector length of NTT is on the order of 2^{20}, rendering such large amount of data unfeasible to be entirely stored into on-chip memory. Of special interest is the four-step NTT version which decomposes the large-scale one-dimension vector into two-dimensional matrix, enabling us to perform independent and parallel sub-NTTs for each row or column. Moreover, this arrangement allows on-chip memory to accommodate several rows or columns of data only, alleviating the pressure on memory usage. Adopting higher dimensional (e.g., five-step) NTT variants can further reduce the size of sub-NTTs, but it will complicate the access pattern, entail more transposition cost and increase the number of multiplications [KKK+22].

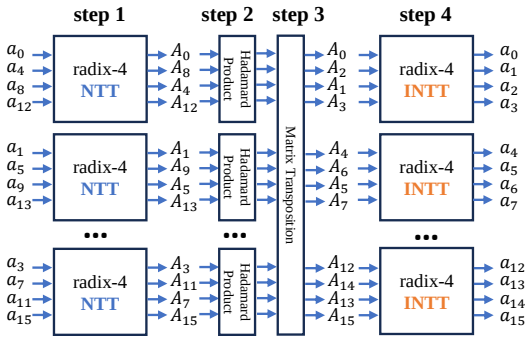


Figure 2: The basic form of four-step 16-point NTT algorithm.

In essence, ZKP-oriented NTT represents a variant of the Fast Fourier Transform (FFT) over the finite field, whose objective is to reduce the complexity of the following matrix-vector multiplication from $\mathcal{O}(N^2)$ to $\mathcal{O}(NlogN)$: $A_i = \sum_{j=0}^{N-1} a_j \cdot \omega_N^{ij} \bmod M, i = 0, 1, ..., N-1$. Here, ω_N is the N-th root of unity over the finite field $\mathbb{F}_M$, whose powers

are also known as the twiddle factors. Previous researchers explore various approaches, such as divide-and-conquer strategy [CT65], index-mapping perspective [Sch96] and tensor technique [TAL93], to derive different variants of FFT, including the mixed-radix, multi-dimension(step) and Good's trick versions and so on. For the sake of clarity, we present the workflow of 16-point four-step NTT in Figure 2, while the concrete index-mapping based derivation process for four-step NTT is shown in Appendix C.

Algorithm 1 Pippenger algorithm [Pip76]

Input: $\vec{s} = (s_0, s_1, ..., s_{n-1})$, with s_i being λ-bit scalar over $\mathbb{F}_{\mathrm{M}}$. Choosing the window size as c-bit. $h = \lceil \frac{\lambda}{c} \rceil$. $s_{i,j} = s_i[j \cdot c : j \cdot c + c - 1]$ is the j-th c-bit chunk of s_i, with $j = 0, 1, ..., h-1$. $\vec{P} = (P_0, P_1, ..., P_{n-1})$, where P_i is the curve point with coordinates over $\mathbb{F}_{\mathrm{Q}}$.

Output: $O = \sum_{i=0}^{n-1} s_i \cdot P_i$.

$O = \mathcal{O}$
for $j = h - 1$ to 0 **do**
  $B = (B_0, B_1, ..., B_{2^c-1}) = (\mathcal{O}, \mathcal{O}, ..., \mathcal{O})$.
  Phase 1. Bucket accumulation
  for $i = 0$ to $n - 1$ **do**
    $ID = s_{i,j}$
    $B_{ID} = PADD(B_{ID}, P_i)$ ▷ $B_t = SUM(P_i | s_{i,j} == t)$
  end for
  Phase 2. Bucket aggregation
  $G_j = \mathcal{O}$, $T_j = \mathcal{O}$
  for $k = 0$ to $2^c - 1$ **do**
    for $m = 2^c - 1$ to k **do**
      $T_j = PADD(T_j, B_m)$
      $G_j = PADD(G_j, T_j)$ ▷ $G_j = \sum_{t=1}^{2^c-1} t \cdot B_t$
    end for
  end for
  Phase 3. Group aggregation
  for $i = 0$ to $c - 1$ **do**
    $O = PADD(O, O)$
  end for
  $O = PADD(O, G_j)$ ▷ $O = 2^{jc} \cdot O + G_j$
end for
return O

2.4 Pippenger Algorithm

The Pippenger algorithm, also called as bucket method [BDLO12], is to MSM what NTT is to polynomial multiplication. It is essentially the variant of distributed arithmetic [BS91] used to efficiently compute the inner product. Recently, numerous variants of Pippenger algorithm are developed to achieve elegant parallelism and computation-memory trade-off [LWY+23, ZHY+24]. But most of them are mainly tailored for the software platforms like CPU and GPU. In our opinion, the basic form of Pippenger algorithm still remains competitive for hardware implementation since it requires less memory footprint, relatively regular controller and no precomputation.

As shown in Algorithm 1, the Pippenger algorithm can be divided into three phases, which essentially perform the distributed arithmetic [BS91] as below:

$$O = \sum_{i=0}^{n-1} s_i \cdot P_i = \sum_{i=0}^{n-1} \sum_{j=0}^{h-1} 2^{jc} s_{i,j} \cdot P_i = \sum_{j=0}^{h-1} 2^{jc} \sum_{i=0}^{n-1} s_{i,j} \cdot P_i = \sum_{j=0}^{h-1} 2^{jc} \cdot G_j \tag{1}$$

At phase 1 named bucket accumulation, each curve point P_i along with scalar chunk $s_{i,j}$ is accordingly accumulated with the point B_t stored in bucket t ($= s_{i,j}$). At phase 2, all the curve points are sorted and compressed into their buckets, so that the bucket aggregation is performed by adding up the weighted points stored in each bucket to obtain G_j. This weighted summation can be specifically handled with the addition chain as shown in line

10-16 of Algorithm 1. At phase 3, each bucket aggregated result G_j weighted by 2^{jc} is iteratively added up to calculate the final result.

2.5 Montgomery Modular Multiplication

The Montgomery modular multiplication can at least date back to [Mon85], whose fundamental principle is to add a specific multiple of modulus $q \cdot M$ to the $w \times w$-bit multiplication result $S = A \cdot B$, so that the half lower part of $S + q \cdot M$ turns out to be zeros. Herein, q is the quotient determined as $q = (S \bmod R) \cdot M' \bmod R$, with R being the radix 2^w and $M' = -M^{-1} \bmod R$. In this way, the final modular reduction is replaced with a simple shift operation, which turns out $S = (S + q \cdot M) >> w \bmod M = A \cdot B \cdot R^{-1} \bmod M$. This intricate reduction method, popularly applied in both hardware and software platforms [Lon23], circumvents the time-consuming division inherent in straightforward reduction method, and could be implemented in a constant-time manner.

Sugar of modulus. The high-radix Montgomery MM, as illustrated in Algorithm 2, is also widely adopted to trade speed with area. At this time, the operand A is divided into several m-bit chunks, with each iteration scanning a single chunk A_i alongside the entire operand B as the input. Accordingly, the bit-width of quotient q is shrunk to m-bit. The NTT-friendly modulus M adheres to the condition $M \bmod 2^n = 1$, with 2^n being the vector length. Thus, the modulus can be expressed as $M = t \cdot 2^n + 1$, which is similar to the Proth number or Montgomery-friendly number apart from the difference that t is a large dense prime number. This specific form lends itself to the high-radix Montgomery reduction method. Because the constant factor is given by $M' = -1 \bmod R$, if the Montgomery radix is chosen as $R = 2^m = 2^n$. Obviously, such parameter selection can reduce one multiplication cost in the iterative computation, *i.e., the calculation of the quotient is almost cost-free:* $q = -S \bmod R$. This optimization trick brings the advantage of reducing both the area overhead and critical path, which is further enhanced when the bit-width of modulus becomes large. Interestingly enough, we also find that the coordinate modulus Q of some pairing-friendly curves, such as BLS12-381, BLS12-377, has the similarly low-cost quotient determination even if it is not the NTT-friendly modulus. Based on our review, this is the first work to unleash the power of specific ZKP moduli by exploring the low-cost quotient calculation.

Algorithm 2 High-radix Montgomery MM [KKAK96]

Input: w-bit multiplicand $A = (A_0, A_1, ..., A_{n-1})_m$, multiplier B, modulus M. radix $R = 2^m$ and $n = \lceil w/m \rceil$, $M' = -M^{-1} \bmod R$.
Output: w-bit $S = A \cdot B \cdot R^{-1} \bmod M$

1: $S^{(0)} = 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad S^{(i)} = S^{(i)} + A_i \cdot B$ ▷ the 1^{st} $m \times w$-bit multiplication.
4: $\quad q_i = (S^{(i)} \bmod R) \cdot M' \bmod R$ ▷ the 2^{nd} $m \times m$-bit modular multiplication.
5: $\quad S^{(i+1)} = (S^{(i)} + q_i \cdot M)/R$ ▷ the 3^{rd} $m \times w$-bit multiplication.
6: **end for**
7: **if** $S^{(n)} > M$ **then**
8: $\quad S = S^{(n)} - M$
9: **end if**
10: **return** S

3 Proposed Dual-precision LBFP MM

3.1 Analysis on Existing Montgomery MM

Identifying the load imbalance. To prevent misunderstandings, we highlight that the *fully-pipelined* property refers to the pipelined data-path without any feedback loop under the context of this work. The extreme case opposite to fully-pipelined design is the

digit-serial style [MR18], with the *partially-pipelined* design located between two extremes. The fully-pipelined Montgomery multipliers in prior works [SNF+19, HMR23, MK22] are implemented by directly unrolling the fixed-radix Montgomery algorithm as shown in Figure 3. However, this architecture presents some deficiencies that remained overlooked for a long time. On one hand, as marked by the red box ① in Figure 3, the multiplication of $A_i \cdot B$ should be completed before the multiples of modulus $q \cdot M$. Such inherent data dependency easily results in a prolonged critical path, necessitating the insertion of numerous pipeline registers to attain high frequency. On the other hand, as marked by the red box ②, to synchronize the operands before the next iteration, approximately $i \cdot d$ pipeline registers need to be inserted for the result of $A_i \cdot B$. This implies that the register overhead gradually increases with iteration count i, thereby resulting in a serious imbalance in the pipeline workload. As the bit-width of the modulus increases, these drawbacks become more pronounced.

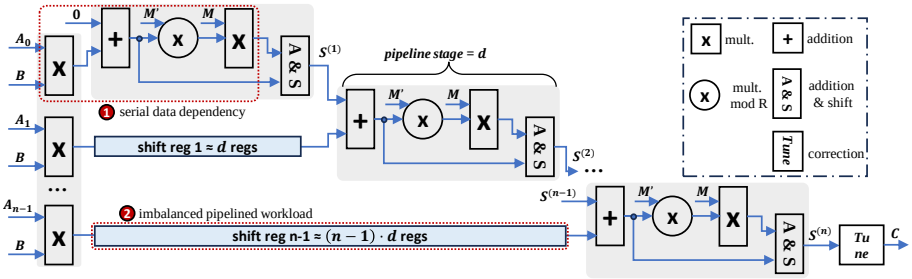


Figure 3: Observations on the typical fully-pipelined Montgomery MM.

Limitations of classic quotient-delayed technique. The data dependency within Algorithm 2 is summarized as three parts. (1) The immediate result $S^{(i)}$ should be computed before the quotient determination $q_i = (S^{(i)} \bmod R) \cdot M' \bmod R$; (2) The quotient determination should precede the multiples of modulus $q_i \cdot M$; (3) The computation of $A_i \cdot B$ is used to update the immediate result $S^{(i)}$. Consequently, the critical path is three multiplications and two additions. Prior work [Oru95] proposes the so-called quotient-delayed technique to decouple the data dependency between computations of intermediate result and quotient. Specifically, this technique utilizes the quotient q_{i-d} (obtained from the $(i-d)$-th iteration) to compute $q_{i-d} \cdot M$, so that $A_i \cdot B$ can be performed in parallel with $q_{i-d} \cdot M$ at the current i-th iteration. However, this technique will increase the iteration count and incur additional post-processing, which are required to adjust the final result back to the range $[0, M)$. Therefore, subsequent works [SNF+19, BRM19] typically choose a delay degree of $d = 1$ to make the added cost minimal. The recent work [ZCP23] presents an alternative algorithm that decouples the data dependency meanwhile avoiding the indefinite number of post-processing, but it appears to double the bit-width of quotient. So far as we know, none of the prior Montgomery algorithms can simultaneously meet the following conditions: (1) The computations of $q \cdot M$ and $A_i \cdot B$ are done in parallel; (2) The final reduction result lies in a constant range without additional post-correction [Oru95]; (3) The bit-width of quotient will not be enlarged [ZCP23]; (4) The iteration count is increased no more than once.

3.2 Proposed Modularized and Efficient Techniques

Improved quotient-decoupled technique. Motivated by the limitations of existing quotient-delayed technique, we propose two new alternative methods that not only break the data dependency but also meet the aforementioned conditions. The first method resorts to only rearranging the order of iterative computations, whereas the second one consists in changing the iterative computation while still maintaining the original iterative

order. For brevity, this work elaborates the principle of the first method. It is observed from the Algorithm 2 that the initial value $S = 0$ and the relationship between $S^{(i+1)}$ and $S^{(i)}$ can be iteratively expressed as:

$$\begin{aligned} &S^{(1)} = (A_0 \cdot B + q_0 \cdot M)/R \\ &S^{(2)} = (A_1 \cdot B + q_1 \cdot M + S^{(1)})/R = (A_1 \cdot B + q_1 \cdot M)/R + (A_0 \cdot B + q_0 \cdot M)/R^2 \\ &... \\ &S^{(i+1)} = (A_{i+1} \cdot B + q_{i+1} \cdot M + S^{(i)})/R \\ &= (A_{i+1} \cdot B + q_{i+1} \cdot M)/R + (A_i \cdot B + q_i \cdot M)/R^2 + ... + (A_0 \cdot B + q_0 \cdot M)/R^{i+1} \end{aligned} \tag{2}$$

Interestingly, we find that if the $q_i \cdot M$ at current iteration is moved to the next iteration, then anther periodical iteration behavior can also be formed as:

$$\begin{aligned} &S^{(0)} = A_0 \cdot B \\ &S^{(1)} = (A_1 \cdot B \cdot R + q_0 \cdot M + S^{(0)})/R = A_1 \cdot B + (A_0 \cdot B + q_0 \cdot M)/R \\ &S^{(2)} = (A_2 \cdot B \cdot R + q_1 \cdot M + S^{(1)})/R = A_2 \cdot B + (A_1 \cdot B + q_1 \cdot M)/R + (A_0 \cdot B + q_0 \cdot M)/R^2 \\ &... \\ &S^{(i+1)} = (A_{i+1} \cdot B \cdot R + q_{i+1} \cdot M + S^{(i)})/R \\ &= A_{i+1} \cdot B + (A_i \cdot B + q_i \cdot M)/R + ... + (A_0 \cdot B + q_0 \cdot M)/R^{i+1} \end{aligned} \tag{3}$$

In this way, the computations of $q_{i-1} \cdot M$ and $A_i \cdot B$ can be performed in parallel. Compared to the previous arts, we reformulate the widely used quotient-delayed technique, crucially capturing the lower bound of iteration count.

Proposed load balancing technique. Although decoupling the quotient determination helps alleviate the load imbalance issue, we further propose a simple yet effective method to augment the profit. Despite its broader generality, the method is analyzed in the context of Montgomery multiplication, since it is prevailing and exhibits strong synergy with modulus from pairing (§2.5). The core idea is in similar fashion to the square root carry select adder that increases the bit-width of sub-adder with carry propagation stages [HCG05]. In this manner, the accumulated carry propagation delay until stage i is balanced by correspondingly amplifying the bit-width of sub-adder at stage i. Hence, the intuitive imitation is to gradually increase the bit-width of sub-multiplication $A_i \cdot B$ with stage i, so that the accumulated pipeline registers can be fully used to pipeline the sub-multiplication, leading to a more balanced pipeline partitioning. However, due to the impact of extra modular reduction, the analogous version is non-trivial. Expanding upon this discovery, the w-bit multiplicand A is divided into a chunk vector based on the mixed-radix $R_i = 2^{r_i}$, and needs to meet the condition: $R_0 \leq R_1 \leq ... \leq R_{n-1}$. Then, we figure out that the matched constant factor for each $A_i \cdot B$ is calculated as $[M_i']_{r_i} = -M^{-1} \bmod R_i$. In the following, we will prove that the Montgomery algorithm is *"radix-friendly"*, which indicates that the iterative computation based on different radices can be linearly assembled to obtain the final result correctly.

Putting it together. The proposed two techniques are orthogonal to each other, and can be combined to generate the new load-balanced Montgomery algorithm with decoupled quotient determination, as shown in Algorithm 3. These two techniques are also modularized and potentially beneficial for the Barret reduction algorithm. The radices R_i are carefully selected to make $[M_i']_{r_i}$ intensively sparse, thereby lowering the multiplication cost for determining the quotient (at line 10 of Algorithm 3). Recalling that the modulus used in zk-SNARK is in form of $M = t \cdot 2^N + 1$, the feasible choice for R_i is the value that can be evenly divided by 2^N or has common dividers with 2^N. Additionally, the truncated addition is also applied to the final addition within each iteration, which is expressed as:

$$\begin{aligned} &S^{(i)} = (S^{(i-1)} + [q_{i-1}]_{r_{i-1}} \cdot M)/R_{i-1} + [A_i]_{r_i} \cdot B \\ &= S^{(i-1)}/R_{i-1} + ([q_{i-1}]_{r_{i-1}} \cdot M)/R_{i-1} + (t\ !=0) + [A_i]_{r_i} \cdot B, \text{ where } t = S^{(i-1)} \bmod R_{i-1} \end{aligned} \tag{4}$$

This shift-before-addition strategy at line 12 of Algorithm 3 helps reduce the width of

addition and lower area cost. To confirm the correctness of two proposed techniques, we make a dedicated proof as below.

Algorithm 3 Proposed Load-balanced Montgomery MM

Input: w-bit multiplicand A, multiplier B, modulus M. Montgomery radix $R = 2^w$, Mixed radix $R_i = 2^{r_i}$, $i = 0, 1, ..., n-1$. Usually, we select: $0 = r_0 \leq r_1 \leq ... \leq r_{n-1}$. $[M'_i]_{r_i} = -M^{-1} \bmod R_i$.
Output: $S = A \cdot B \cdot R^{-1} \bmod M$.

1: *Perform preprocess:*
2: Defining the Montgomery mixed radix as $R = 2^w = (r_0, r_1, ..., r_{n-1})$, $R_i = 2^{r_i}$ $(0 \leq i \leq n-1)$. Then, we have: $w = \sum_{i=0}^{n-1} r_i$, $R = \prod_{i=0}^{n-1} R_i$.
3: Express the w-bit multiplicand A with mixed-radix form: $A = ([A_0]_{r_0}, [A_1]_{r_1}, ..., [A_{n-1}]_{r_{n-1}})$. Then, we have: $0 \leq [A_i]_{r_i} \leq R_i - 1$, $A = \sum_{k=0}^{n-1}([A_k]_{r_k} \cdot \prod_{j=0}^{k} R_j)$.
4: *Perform Montgomery modular multiplication:*
5: /*`Initialization`*/
6: $S^{(0)} = [A_0]_{r_0} \cdot B$
7: /*`Iteration`*/
8: **for** $i = 1$ to $n-1$ **do**
9: $\quad t = S^{(i-1)} \bmod R_i$
10: $\quad [q_{i-1}]_{r_{i-1}} = t \cdot [M'_{i-1}]_{r_{i-1}} \bmod R_{i-1}$
11: $\quad Z = t == 0\ ?\ 0 : 1$ ▷ For reducing the bit-width of final addition.
12: $\quad S^{(i)} = S^{(i-1)}/R_{i-1} + [A_i]_{r_i} \cdot B + ([q_{i-1}]_{r_{i-1}} \cdot M)/R_{i-1} + Z$ ▷ Two parallel multipliers.
13: **end for**
14: /*`The last iteration`*/
15: $t = S^{(n-1)} \bmod R_{n-1}$
16: $[q_{n-1}]_{r_{n-1}} = t \cdot [M'_{n-1}]_{r_{n-1}} \bmod R_{n-1}$
17: $Z = t == 0\ ?\ 0 : 1$
18: $S^{(n)} = S^{(n-1)}/R_{n-1} + ([q_{n-1}]_{r_{n-1}} \cdot M)/R_{n-1} + Z$ ▷ Leave out $[A_n]_{r_n} \cdot B = 0$.
19: **if** $S^{(n)} > M$ **then**
20: $\quad S = S^{(n)} - M$
21: **end if**
22: **return** S

Theorem 1. *Algorithm 3 is a new variant of quotient-decoupled load-balanced Montgomery modular multiplication that computes $S = A \cdot B \cdot R^{-1} \bmod M$ without increasing the iteration count and incurring additional correction.*

Proof. We could start from the iteration at line 12 of Algorithm 3 to seek the relation between $S^{(i)}$ and $S^{(i-1)}$. First, the equation at line 12 is restored to its orginal form: $S^{(i)} = (S^{(i-1)} + [q_{i-1}]_{r_{i-1}} \cdot M)/R_{i-1} + [A_i]_{r_i} \cdot B$. Then, we unfold the iteration until expressing $S^{(i)}$ with $S^{(0)}$:

$$\begin{aligned}
&S^{(i)} \cdot R_{i-1} = S^{(i-1)} + [q_{i-1}]_{r_{i-1}} \cdot M + A_i \cdot B \cdot R_{i-1} \\
&\Rightarrow S^{(i)} \cdot R_{i-1} \cdot R_{i-2} = A_i \cdot B \cdot R_{i-1} \cdot R_{i-2} + [q_{i-1}]_{r_{i-1}} \cdot R_{i-2} \cdot M + S^{(i-1)} \cdot R_{i-2} \\
&= A_i \cdot B \cdot R_{i-1} \cdot R_{i-2} + [q_{i-1}]_{r_{i-1}} \cdot R_{i-2} \cdot M + A_{i-1} \cdot B \cdot R_{i-2} + [q_{i-2}]_{r_{i-2}} \cdot M + S^{(i-2)} \\
&= (A_i \cdot R_{i-1} \cdot R_{i-2} + A_{i-1} \cdot R_{i-2}) \cdot B + ([q_{i-1}]_{r_{i-1}} \cdot R_{i-1} + [q_{i-2}]_{r_{i-2}}) \cdot M + S^{(i-2)} \\
&... \\
&\Rightarrow S^{(i)} \cdot \prod_{j=1}^{i} R_{j-1} = B \cdot \sum_{k=1}^{i}([A_k]_{r_k} \cdot \prod_{j=1}^{k} R_{j-1}) + M \cdot \sum_{k=1}^{i}([q_{k-1}]_{r_{k-1}} \cdot \prod_{j=0}^{k-1} R_j) + S^{(0)}
\end{aligned} \tag{5}$$

Note that $S^{(0)}$ is initialized as $[A_0]_{r_0} \cdot B$. The last iteration (line 15 to 18 of algorithm 3) leaves out the accumulation with $[A_n]_{r_n} \cdot B$ since $[A_n]_{r_n} = 0$. Considering the penultimate

iteration to assign $i = n-1$, we have $A = \sum_{k=0}^{n-1}([A_k]_{r_k} \cdot \prod_{j=0}^{k} R_j)$. Defining $q = \sum_{k=0}^{n-1}([q_k]_{r_k} \cdot \prod_{j=0}^{k} R_j)$, we have:

$$\begin{aligned} & S^{(n-1)} \cdot \prod_{j=1}^{n-1} R_{j-1} = B \cdot \sum_{k=0}^{n-1}([A_k]_{r_k} \cdot \prod_{j=0}^{k} R_j) + B \cdot \sum_{k=0}^{n-2}([q_k]_{r_k} \cdot \prod_{j=0}^{k} R_j) \\ & \Rightarrow S^{(n)} \cdot \prod_{j=0}^{n-1} R_j = [q_{n-1}]_{r_{n-1}} \cdot M \cdot R_{n-1} + S^{(n-1)} \cdot \prod_{j=1}^{n-1} R_{j-1} \\ & = B \cdot \sum_{k=0}^{n-1}([A_k]_{r_k} \cdot \prod_{j=0}^{k} R_j) + M \cdot \sum_{k=0}^{n-1}([q_k]_{r_k} \cdot \prod_{j=0}^{k} R_j) = A \cdot B + q \cdot M \\ & \Rightarrow S^{(n)} \cdot R = A \cdot B + q \cdot M \end{aligned} \tag{6}$$

Based on the Bézout's identity $R \cdot (R^{-1} \bmod M) + M \cdot (M' \bmod R) = 1$ and equation 6, we can derive the final result: $S = A \cdot B \cdot R^{-1} \bmod M$. □

3.3 Case Study: Dual-precision MM for BLS12-381 Curve

We take the typical BLS12-381 curve parameter as an example to instantiate a specific LBFP modular multiplier that supports dual precisions for both NTT and MSM. The optimization strategies can also be adapted to other parameters such as the BLS12-377 curve. Here, we choose the mixed radices as: $R_0 = R_1 = 2^{32}, R_2 = 2^{48}, R_3 = 2^{64}, R_4 = 2^{80}, R_5 = 2^{128}$, and we further have: $R_a = \prod_{i=0}^{4} R_i = 2^{256}, R_b = \prod_{i=0}^{5} R_i = 2^{384}$. Then, the constant factors under different radices are derived and summarized in Table 7. To support double-precision MM, we implement the 384-bit modular multiplier that is naturally compatible with 256-bit MM, albeit with some redundancy. Although a 128-bit modular multiplier based on CIOS method can be iteratively utilized to achieve 256-bit and 384-bit MM without redundancy, this approach introduces feedback loops in the data-path, failing to meet the requirements for a fully-pipelined design. Collectively, we also integrate other orthogonal ideas to further enhance the performance of this MM, which are listed as below.

- **Customizing the constant multiplication.** To unleash the power of the special modulus, we implement the quotient determination $[q_i]_{r_i} = (S^{(i)} \bmod R_i) \cdot [M_i']_{r_i} \bmod R_i$ (at line 10/16 of Algorithm 3) with special multiplier (SM) comprised of shift and addition operations, consequently lowering the area cost. We also represent the moduli of NTT and MSM in non-adjacent form (NAF) to reduce the computational intensity of $q_i \cdot M$ based on the constant multipliers (CM).

- **Truncated LSB/MSB multiplication.** The truncated LSB multiplication refers to the multiplication for quotient calculation where only the lower part is taken as the result. This approach helps reduce area overhead by leaving out the higher-part multiplication [DL18], especially for dense constant factors. The truncated MSB multiplication indicates the situation where the lower part of $S + q \cdot M$ will be zeros, allowing for opportunities to omit partial parts of multiplication. However, directly disregarding the lower bits of M will introduce unpredictable calculation errors [DL18][LP21]. Here, we adopt a different approach [Oru95] based on the fact that the lower r_i bits of $[M_i']_{r_i} \cdot M + 1$ will be zeros. Thus, the computation of $(q \cdot M) >> r_i$ can be replaced with $S_0 \cdot ([M_i']_{r_i} \cdot M >> r_i) + S_0 - 1$ in that $q = S_0 \cdot [M_i']_{r_i} \bmod r_i$. In other words, the SM unit for quotient determination and CM for $q \cdot M$ are merged to a single truncated multiplier, which is especially beneficial for the cases concerning dense constant factors.

- **Karatsuba.** For 128 × 128-bit and 80 × 80-bit sub-multiplications within the 128 × 384-bit and 80 × 384-bit multiplications, we further adopt the well-known

karatsuba technique to generate partial products and thus reduce the area overhead. This is due to the fact that at certain threshold of bit-width Karatsuba starts to outperform schoolbook algorithms in terms of area-efficiency.

The detailed dual-precision LBFP MM for BLS12-381 curve is shown in Figure 4, which takes 9 clock cycles to obtain the final result. Here, the SM/CM units marked by the blue color are implemented by two separated constant multipliers to be selected for 256-bit and 384-bit modulus, respectively. According to the synthesis report under TSMC 28nm process, the critical path consists of a 32 × 384-bit multiplier, SM0, CM0 and a 416-bit adder within the part I and part II.

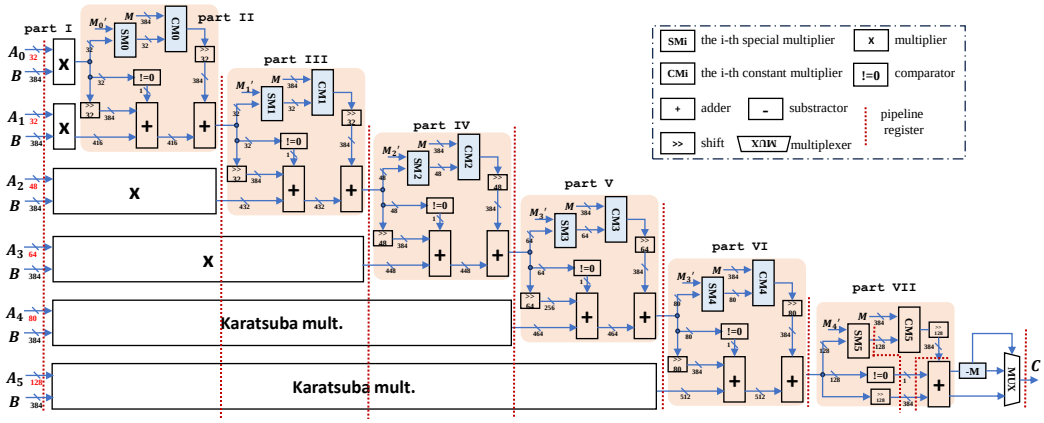


Figure 4: An instantiation of dual-precision LBFP MM for BLS12-381 curve.

4 Unified Accelerator for NTT & MSM

Taking BLS12-381 curve as an example, we will introduce the high-level architecture and configuration for NTT and MSM first. Then, the optimization strategies, timing scheduling mechanism and reasons behind the choice of design parameters will be covered later for NTT and MSM modes, respectively.

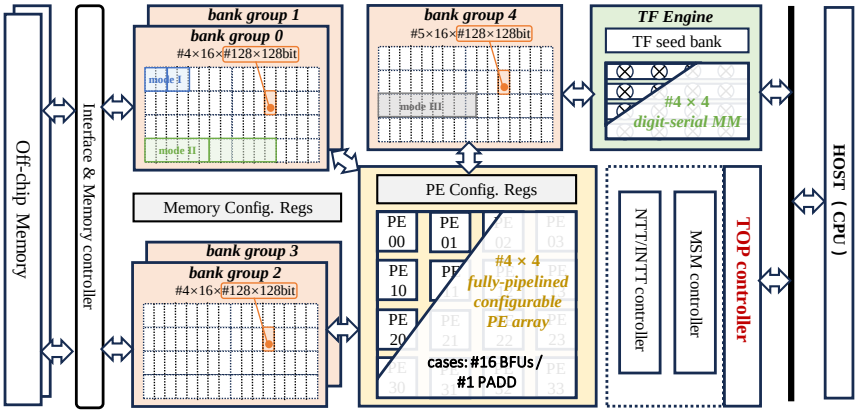


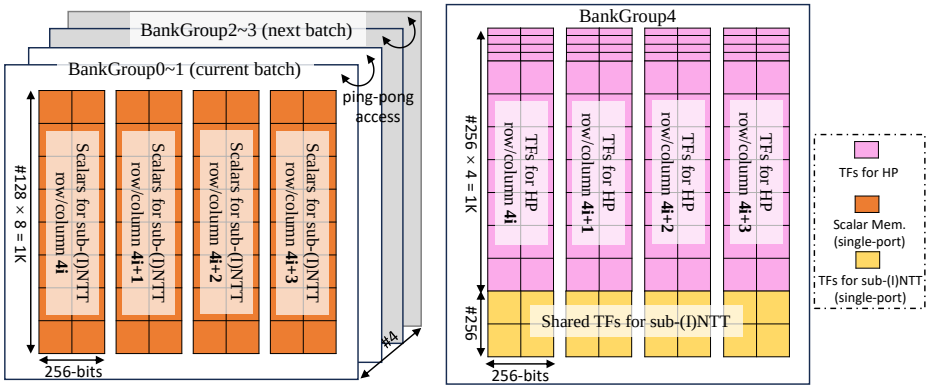
Figure 5: The unified architecture for NTT/INTT and MSM.

4.1 High-level Architecture Overview

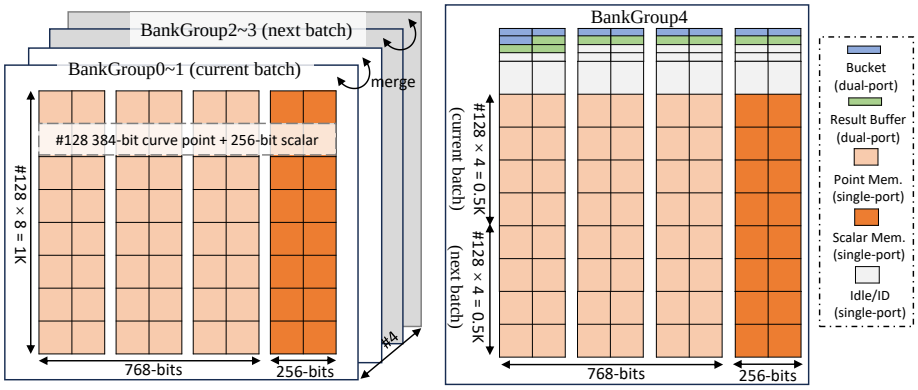
Figure 5 presents the unified architecture for both NTT/INTT and MSM, which mainly consists of five bank groups, a twiddle factor (TF) engine, a configurable processing element (PE) array, three hierarchical controllers and two ping-pong off-chip memories. At the

initial phase, data points to be processed by NTT and MSM are loaded from the host CPU to the off-chip memory. Particularly, the DRAM layout for NTT is customized as the tiled data blocks shown in Appendix B, which is conducive to minimizing the row buffer miss that arises from large address stride when accessing rows and columns of large-scale matrix [AFH16]. Besides, the classic four-step NTT requires extra cycle consumption and transpose unit, which takes up about 14% of the area per compute cluster [SFK+21]. Fortunately, the matrix transposition can be eliminated in our design, since the customized layout already supports both row-wise and column-wise access efficiently. Then, the mode signals and configuration context are sent from the host to prepare for the next execution.

Memory organization. In general, each memory group is divided into 64/80 banks, with each bank equipped with 128×128-bit memory cells. In this way, two, six or nine memory cells can be assembled to accommodate a 256-bit scalar, a 768-bit curve point with projective coordinate ($Z = 1$) or a 1152-bit curve point ($Z \neq 1$). Thus, these bank groups under mode I - III can be shared to support both NTT and MSM, ultimately improving the resource utilization. Additionally, partial banks within bank group 4 are dual-port SRAMs used as buckets and result buffer for MSM, such that two access requests during the bucket accumulation step can be met without pipeline stalls. In summary, the memory partition strategies for NTT and MSM modes are depicted in Figure 6, which will be further referred and elaborated in the following sections.



(a) NTT Mode: Memory partition for 2^{20} NTT (can be extended to $2^{22}, 2^{24}$)



(b) MSM Mode: Memory partition for 2.5K on-chip MSM

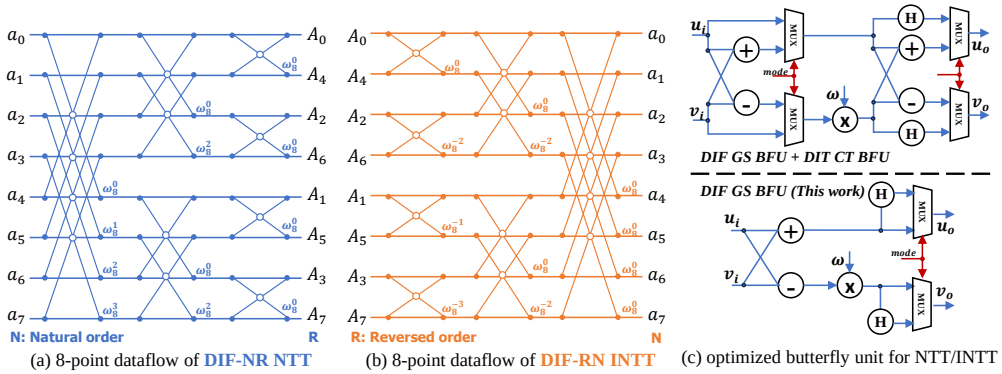
Figure 6: Proposed memory partition strategies for NTT and MSM.

PE array arrangement. To support all three steps of MSM, the point adder unit adopts the complete point addition formula [BL95, RCB16] instead of the mixed addition one [RDQY24]. Because the latter cannot be performed between two curve points both

with projective coordinates, which are likely to occur during bucket/group aggregation phases. It takes 12 384-bit modular multipliers and approximately 20 modular adders to implement the complete point addition formula. We also adopt the Gentleman Sande (GS) butterfly unit (BFU) for both NTT and INTT, which occupies one 256-bit modular multiplier, adder, subtractor and half unit. As a result, the 4×4 PE array is devised to support the configuration for 16 BFUs and 1 point adder unit. Each PE is composed of one modular multiplier, adder, subtractor and several multiplexers. Among them, 12 PEs are built with 384-bit data channel while the remaining 4 PEs contain the 256-bit data channel. Since the Montgomery radix is chosen as $R \geq 2M$, the modular adder can be replaced with reduction-free adder to further save area cost like the lazy reduction technique [CDF+11].

4.2 Optimization Strategies for NTT Mode

Limitations of existing designs. Prior works (e.g., [ZWZ+21]) adopt the SDF pipelined sub-NTT architecture to take in one data point per cycle. This streamlining architecture aims to mitigate the requirement of off-chip memory bandwidth and avoid the multi-bank access conflicts. However, the typical SDF pipelined architecture only achieves 50% hardware utilization rate, and the segmented FIFOs are hard to be reused by MSM. [HMR23] points out that the SDF NTT architecture takes up approximately $2n + logn \cdot l$ cycles to process n data points when the butterfly operation consumes l cycles. If an equal number (i.e., $logn$) of BFU is utilized, the multi-bank NTT architecture consumes $n/(2logn) \cdot logn + l = n/2 + l$ clock cycles only. Since the parameters n and l are particularly large within zk-SNARK, the cycle count for the SDF type far exceeds that of multi-bank based architecture. Another similar pipelined NTT architecture is based on the Multi-Path Delay Commutator (MDC) technique, which achieves 100% hardware utilization rate but still consumes $n + logn \cdot l$ clock cycles. Hence, this work pivots towards the scalable multi-bank NTT architecture, which also facilitates the hiding of the off-chip memory access latency by adjusting the number of parallel BFUs.



(a) 8-point dataflow of DIF-NR NTT (b) 8-point dataflow of DIF-RN INTT (c) optimized butterfly unit for NTT/INTT

Figure 7: An example of rearranged dataflow for 8-point NTT/INTT.

Algorithmic optimizations. The complete four-step NTT/INTT, as shown in Algorithm 4, is optimized from two-fold aspects. (1) Avoiding bit-reversed issue and simplifying the unified BFU. Although conducting bit-reversing for addresses is essentially a low-cost rewiring operation, varying vector lengths involve different rewiring channels, thus amplifying the overhead of multiplexers. Hence, it is advisable to remove the bit-reversed overhead through algorithmic improvement. By studying literature works, we find that most prior works adopt the DIF-NR INTT algorithm, whereas the DIF-RN INTT is fewly discussed in detail. By adopting similar techniques to [CYY+22], this work proposes the paired DIF-RN INTT and DIF-NR NTT based on GS BFU only, which

simplifies the unified BFU (Figure 7 (c)) by reducing one modular adder, subtractor and two multiplexers. (2) Based on the property: $\omega_n^{-i} = \omega_n^{n-i} = \omega_n^{n/2} \cdot \omega_n^{n/2-i} = -\omega_n^{n/2-i}$, we can reuse the TFs of sub-NTT for sub-INTT. As shown in Algorithm 5 & 6, we ultimately need to store $n/2-1$ TFs for performing both n-point sub-NTT and sub-INTT, whose 8-point data flows are depicted in Figure 7 (a) & (b), respectively.

Proposed sub-NTT kernel. As shown in Figure 8, each sub-NTT kernel serves one row/column of matrix, consisting of address generators, ping-pong banks, shuffles, and 4 parallel BFUs. Four sub-NTT kernels share the same TF engine and TF memory, independently conducting four row/column-wise (I)NTTs. As explained in Figure 6 (a), every bank group0-3 is further decomposed into #4×8 independent banks to match the parallel butterfly computations in a ping-pong access mechanism. Inspired by the work [Kun85], we devise an extremely lightweight shuffle to achieve conflict-free memory access for each stage of parallel butterfly computations. Figure 8 depicts the exemplary shuffle for the 0-th stage of 16-point NTT/INTT with 4 BFUs. The two-layer shuffle0 manages the data vector fetched from the bank group0, whereas the two-layer shuffle1 reorders the output results from BFUs and feeds them into the bank group1. Readers can verify that the shuffle structure utilized at the i-th stage ($i > 0$) remains identical to that of the 0-th stage. Unlike the heavy permutation networks in [CYY+22, XHY+20], this shuffle consists of only two types of topology at each layer, maintaining constant two fan-ins for each multiplexer regardless of parallel degree. It can also be extended to support a variable number of BFUs, resulting in a substantial reduction in the interconnection overhead.

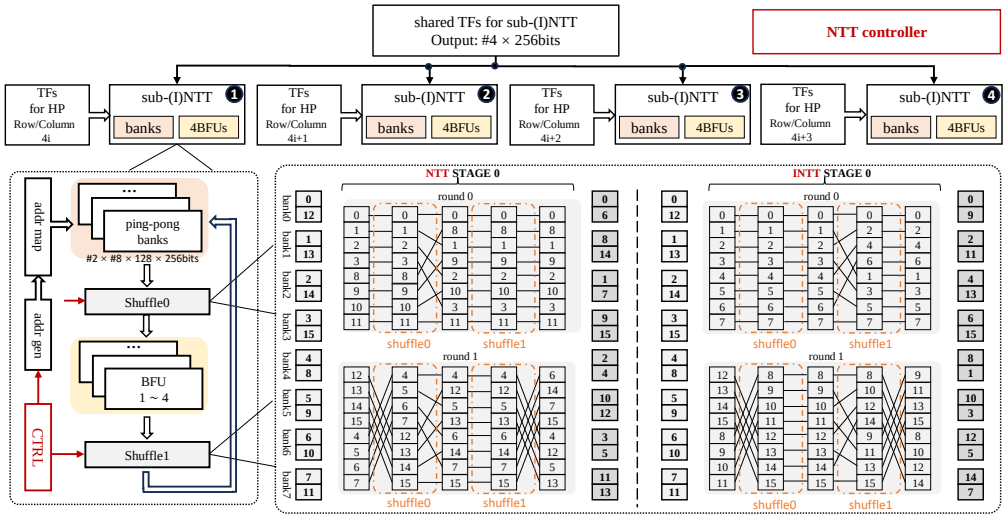


Figure 8: The sub-NTT kernel with exemplary shuffle for 16-point NTT/INTT & 4 BFUs.

Versatile scheduling based on ping-pong banks. Subsequent to and inspired by the versatile strategy in [AFH16], we adopt the ping-pong architecture to overlap the off-chip memory access with the on-chip NTT computation. Specifically, when the on-chip computing units process the current batch of data, the next batch of data can be concurrently prefetched from the off-chip memory to the on-chip unused memory banks. Figure 9 visualizes the tasks of each component when processing four batches of data points for the four-step NTT. Based on the lightweight shuffle mentioned before, we could adjust the parallel degree (i.e., the number of BFUs) to match the on-chip computation with the off-chip memory access latency. Evidently, the benefit of timing schedule based on ping-pong banks is embodied by the following aspects: (1) When the on-chip NTT cluster processes the batch of data d_0, DRAM0 concurrently loads the batch of data d_1 into bank group2; (2) When the on-chip NTT cluster finishes processing the batch of data d_0, it loads the results from bank group1 into DRAM1. At the same time, DRAM0

loads the next batch of data d_2 into the bank group0; (3) When data transfer and NTT computation are under progress, the hardware engine for TFs will work in parallel and should be completed before starting the Hadamard product (HP). The aforementioned timing schedule entirely hides the off-chip memory access latency, considerably reducing the overall execution time. In this way, the cycle count for processing $N \times N$-point four step NTT can be approximately calculated as:

$$\underbrace{N/4 \cdot (N/8 \cdot log_2 N + N/4)}_{\text{processing N-row NTT and HP}} + \underbrace{N/4 \cdot N/8 \cdot (log_2 N)}_{\text{processing N-column NTT}} + \underbrace{\mathcal{O}(N)}_{\text{setup \& end}} \tag{7}$$

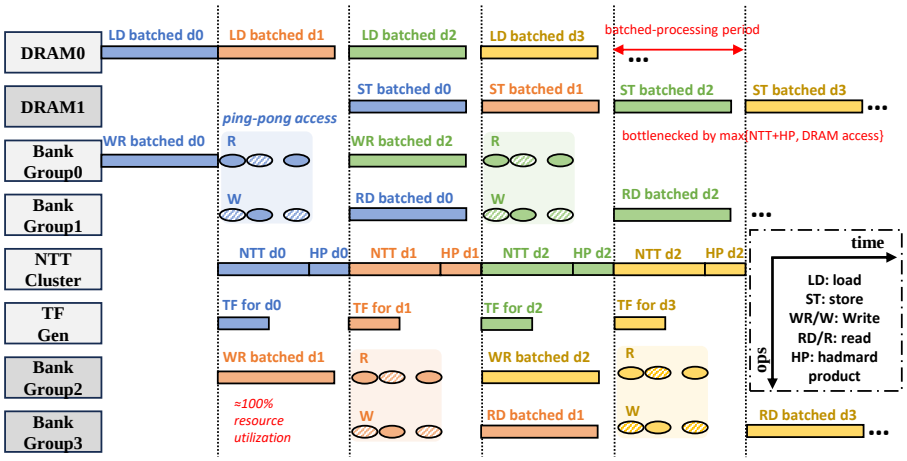
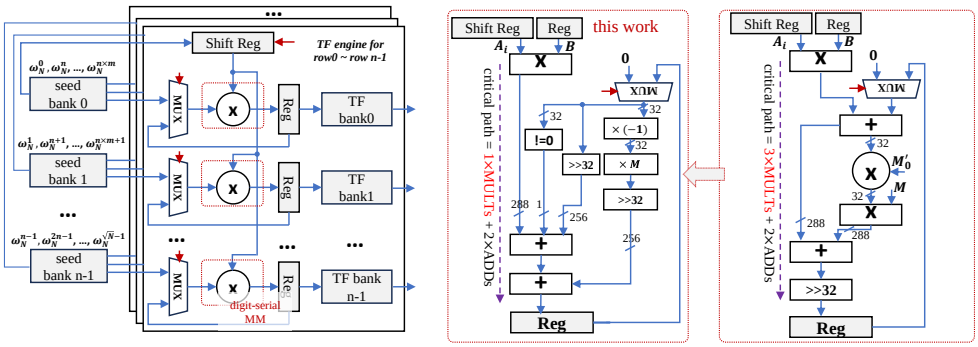


Figure 9: NTT timing schedule for off-chip memory access and on-chip computation.

TF engine. An indispensable operation of four-step NTT is to conduct the HP between each row/column-wise vector and TFs $\omega_{N^2}^{ij}$ $(i = 0, 1, ..., N-1,\ j = 0, 1, ..., N-1)$. Storing all $(N-1)^2$ TFs into on-chip memory would incur tremendous area budget, which motivates us to devise the on-the-fly computation engine as shown in Figure 10. Owing to the property: $\omega_{N^2}^{-i} = \omega_{N^2}^{N^2/2} \cdot \omega_{N^2}^{N^2/2-i} = -\omega_{N^2}^{N^2/2-i}$, we just need to store N seeds $\omega_{N^2}^{i}$ $(i = 0, 1, ..., N-1)$ to generate all the TFs required by the HPs of both NTT and INTT. Four parallel generation engines based on the common seed memory serve the HPs of four rows/columns. Particularly, each engine is composed of 4 digit-serial Montgomery MMs and 4 data banks, which aims to guarantee all necessary TFs are generated before the start of HPs. As a notable optimization, the digit-serial MM is also designed with an instantiation of Algorithm 3, i.e., quotient-decoupled radix-32 Montgomery algorithm, which reduces the critical path by two multiplications compared to the classic method.



(a) Generation engine for twiddle factors (used in the 2rd step) (b) Optimized radix-32 Montgomery modular multiplier

Figure 10: Hardware engine for HP's TFs via the optimized MM.

4.3 Optimization Strategies for MSM Mode

As an operation to aggregate the polynomial with curve point vector, MSM typically makes up a large majority of the workload within zk-SNARK. Different from the large-scale NTT operation that frequently interacts with the off-chip memory, the performance of MSM is dominantly bottlenecked by the massive on-chip point additions. To devise a fully-pipelined hardware architecture for Pippenger algorithm is particularly central to the throughput gain. One of the main concerns for fully-pipelined MSM design is the complicated controller to avoid colliding points and improve pipeline utilization. In the following, we delve into the critical details of the specific architecture, scheduling mechanism, and united point adder design.

Proposed MSM architecture. Considering that the number of curve points assigned to each bucket is related to the specific value of scalars, the workloads among buckets will be unpredictable and possibly imbalanced. Figure 12 depicts the fully-pipelined MSM architecture based on the shared point adder unit, which utilizes a sophisticated scheduler to balance the workload without access conflicts. Compared to the MSM kernel in prior works, the presented architecture fully reuses the memory banks from NTT by manipulating the memory partition. At this time, two bank groups 0-1 in Figure 5 are arranged as #2K×256-bit scalar memory for storing scalar vector and #2K×768-bit point memory for storing curve point vectors fetched from off-chip memory, respectively. Another two bank groups 2-3 are configured as the same manner, which are used to buffer the next batch of scalar/point vectors fetched in parallel with the current MSM computation, thus almost eliminating the DRAM access latency. Bank group 4 is generally split into three parts, functioning as two batches of #0.5K×768-bit point memory, #0.5K×256-bit scalar memory, buckets and result buffers, respectively. Each memory region is paired with a status reg to indicate the full or empty state. Figure 6 (b) portrays the proposed memory partition strategy for MSM in detail.

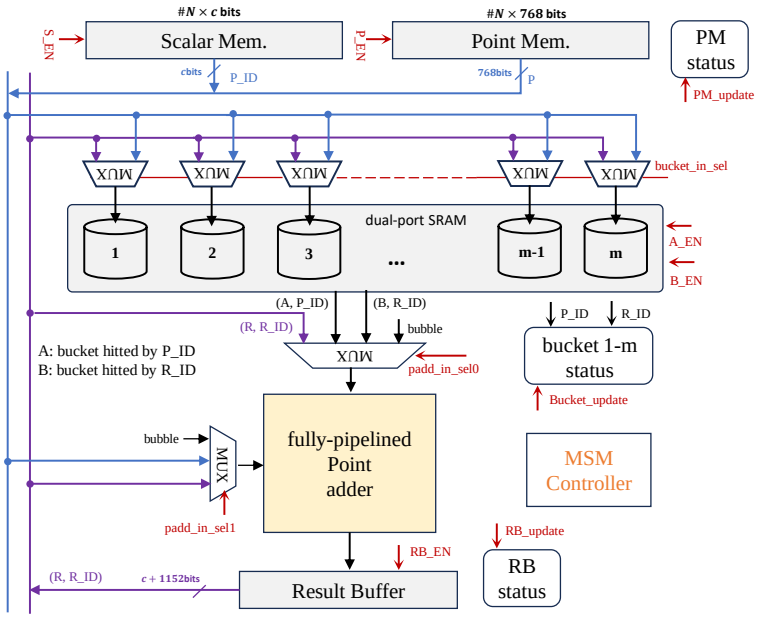
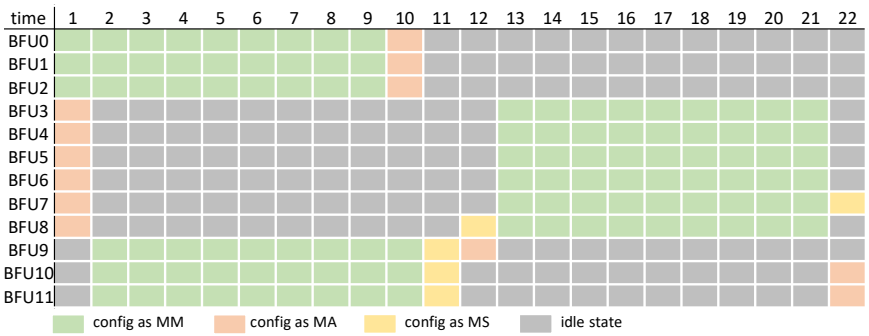


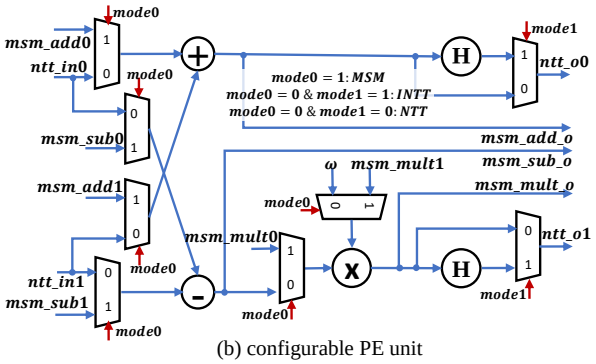
Figure 11: The proposed fully-pipelined MSM architecture.

Compact point adder design. Previous works explore many variants of point addition formula [BL, HWCD08] to reduce the number of required modular multipliers, such

as the mixed addition formula under scaled twisted Edwards curve [RDQY24] consuming 7 modular multipliers and 6 modular adders merely. However, these refined formulas pose certain restrictions to the format of input curve points, which make them inappropriate to the bucket/group aggregation stages. It also employs the extended projective coordinates that consume more memory footprint than the complete point addition formula, which would be a good trade-off on the memory-rich FPGA platform only. Since our design mainly targets at the ASIC implementation, the saved area for the modular multipliers within a single PADD unit would be offset by the largely increased on/off-chip memory overhead for every point. Therefore, we opt for implementing the complete point addition formula [RCB16, BL95] by successfully reusing 12 BFUs from NTT. Figure 12 (a) depicts the spatial and temporal mapping strategy for the point adder with 22 pipeline stages, wherein the modular multiplier (MM) occupies 9 cycles, and modular adder/subtractor (MA/MS) takes up 1 cycle. As can be seen, each PE (BFU) is configured as at most one type of function per cycle, and can not be configured as the same function twice during the 22 pipeline cycles. Using the multiplexers shown in Figure 12 (b), the proposed configurable PE unit can accomplish both BFU and PADD functions without spatial and temporal conflicts. For brevity, the concrete fully-pipelined data-flow and interconnection of the point adder is depicted in Appendix A. Note that the transformation from affine coordinates to projective coordinates in complete formula [RCB16] is cost-free by directly setting $Z = 1$, whereas the twisted formula [HWCD08] requires heavy precomputation to obtain the extended coordinates.



(a) spatial and temporal mapping from #12 BFUs to #1 PADD



(b) configurable PE unit

Figure 12: The mapping from 12 BFUs to 1 PADD based on the configurable PE unit.

Load-aware arbitration mechanism. Compared to the computations for bucket/group aggregation, the bucket accumulation stage is more time-consuming and complex, which is the optimization focus of this work. In contrast to the MSM kernel within PipeZK, the proposed one requires no input FIFOs for the PADD unit by employing a different workload dispatching mechanism. We also adopt the offset binary coding technique to halve the

size of buckets [CA07, LFG23]. Since the input sources for the point adder possibly come from the point memory (PM), buckets and result buffer (RB), it is important to devise an elegant arbiter to circumvent the access conflict and improve the resource utilization. First, the batched scalar and point vector are loaded from DRAM to the on-chip memory. Then, at each cycle, the MSM controller attempts to dispatch one curve point P_i along with one slice of scalar P_ID $= s_{i,j}$ to the bucket with index equal to P_ID. Specifically, the determination of the input source for PADD unit is further divided into ten fine-grained cases shown in Table 1. In general, the input source for PADD unit relies on the status of PM, RB and buckets. For example, case 0-a/b occurs at the initial phase when the first result of pipelined PADD unit is yet streamed out. The PADD input is determined by the status of bucket A. Case 1 indicates that if the index of new point (P_ID) is identical to that of PADD result (R_ID) at current cycle, then they are paired together as the input for PADD unit. Otherwise, we proceed to case 2a-d for further arbitration. In this way, once the operands are available, they are immediately sent to the PADD unit without conflict and pipeline stall, thereby minimizing the occurrence of bubble. The next batched data will be handled only after the current slice vector is scanned and processed based on the above scheduling mechanism. For the sake of clarity, Figure 17 in Appendix A presents an exemplary timing diagram for the one-slice bucket accumulation in detail, which is further divided into three fine-grained phases.

Table 1: The control strategy to manage the input source of PADD unit.

Status	Point Mem	Result Buffer	P_ID == R_ID ?	Bucket A	Bucket B	PADD Inputs	A_EN	B_EN
case 0a	Not Empty	Empty	–	Empty	–	bubble	1	0
case 0b				Full	–	(P,A)	0	0
case 1	Not Empty	Not Empty	Yes	–	–	(P,R)	0	0
case 2a			No	Full	Empty	(P,A)	0	1
case 2b				Empty	Full	(R,B)	1	0
case 2c				Empty	Empty	bubble	1	1
case 2d				Full	Full	(P,A)	0	0
case 3a	Empty	Not Empty	–	–	Empty	bubble	0	1
case 3b				–	Full	(R,B)	0	0
case 4	Empty	Empty	–	–	–	bubble	0	0

NOTES: R: results of PADD. P: curve points from PM. R_ID: index of result R. P_ID: index of P.

Table 2: The parameter configuration for DDR4.

DDR4 type	Capacity	Row	Column	Bank	Bank group	Rank
MT40A256M16GE-062E	4Gb, ×16	15	10	2	1	1
Memory address map	**Data width**	**Freq.**	**Burst length**		**t_{RCD}**	**t_{RP}**
Row-Column-Bank	64 Bytes	3.2 GHz	8		13.75 ns	13.75 ns

5 Implementation Results and Comparisons

In this section, we conduct the case study on NTT and MSM hardware implementation that supports processing data points taken from the BLS12-381 curve. The entire hardware architecture is coded and simulated with Verilog HDL, plus an additional validation of functionality using a cycle-accurate Python model. Under TSMC 28nm HPC process, we synthesize the whole design based on Synopsys Design Compiler P-2019.03. A fast and accurate simulator of DDR named Ramulator [LTB+23] is applied to evaluate the off-chip

memory access, which is also adopted in many relevant works [ZWZ+21] under the similar parameter configuration as shown in Table 2. We also utilize the memory interface IP on the FPGA to verify the simulation data from Ramulator, whose vivado project is available on the given website. The on-chip hardware architecture reserves data/address ports and corresponding control signals for the off-chip memory. Below, we make both theoretical analysis and practical implementation evaluations on the LBFP modular multiplier [1] and overall unified architecture.

5.1 Evaluations on LBFP Modular Multiplier

Area cost of 256/384-bit MM. To quantify the impact of load-balancing (LB) technique on the area savings, we synthesize the 256/384-bit MMs under different strategies shown in Figure 13. Aligning with the theoretical analysis, the LB technique helps reduce the overall area cost by $1.1\times \sim 1.8\times$ when they are staffed with the same througphut.

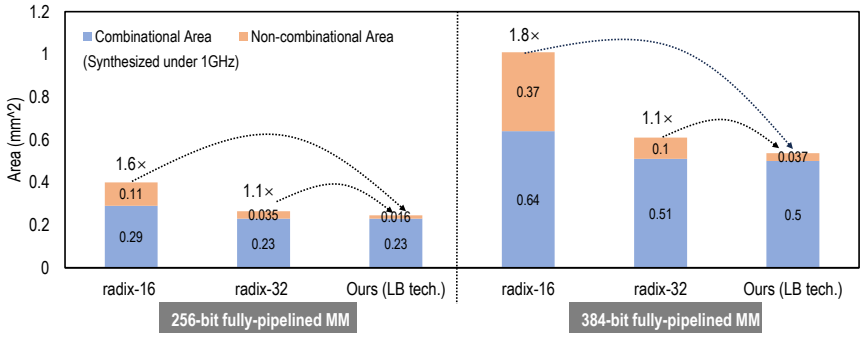


Figure 13: Evaluations on the LB technique for Montgomery MM.

Table 3: The theoretical analysis about different Montgomery reduction methods.

schemes	critical path	# Mult.	quotient decoupled	load balanced	CM. opt.	iteration count	ex. cor.
[KKAK96]	3 mult + 2 add	3	N	N	N	n	N
[Oru95]	1 mult + 2 add	3	Y	N	N	n + d	Y
[MLPJ13]	1 mult + 2 add	3	Y	N	N	n + 3	Y
[SNF+19]	1 mult + 2 add	3	Y	N	N	n + 2	Y
[BRM19]	1 mult + 2 add	3	Y	N	N	n + 3	Y
[ZCP23]	1 mult + 2 add	2	Y	N	Y	n + 8	N
[HMR23]	2 mult + 2 add	2	N	N	Y	n	N
This work	1 mult + 2 add	2	Y	Y	Y	n	N

NOTES: CM. opt. - constant multiplication optimization. ex. cor. - extra correction.

Comparisons about the algorithmic complexity. Table 3 lists some typical and recently-emerged works that utilize different variants of high-radix Montgomery algorithms, accompanied with analysis and comparisons from considerate aspects. Herein, the critical path indicates the longest combinatorial logic path in a single iteration. The classic Montgomery algorithm [KKAK96] requires the lowest iteration count and needs no extra correction to adjust the result back to the modulus range. Nonetheless, it suffers from the largest critical path and highest number of multiplication. In contrast, the classic quotient-delayed Montgomery algorithm [Oru95] exhibits the shortest critical path, at the expense of increased iteration count and extra correction. The works [MLPJ13] [SNF+19] [BRM19] adopt the algorithm from [Oru95] by specifying the delay degree (e.g., $d = 1$ or

[1] Our implementation code is available on https://github.com/xiang-rc/UniNM-Acc.

2). These works respectively devise the scalable systolic array and digit-serial architecture to implement the Montgomery modular multipliers, which also share similar drawbacks with the original algorithm [Oru95]. The recent work [ZCP23] introduces another approach that decouples the quotient computation without extra correction but still encounters the increment of iteration count and doubles the bitwidth of quotient. [HMR23] saves one multiplication cost by utilizing the property of NTT friendly modulus while having similar shortcomings with [KKAK96]. In a nutshell, the LBFP MM integrates quotient-delayed, mixed-radix representation and constant multiplication optimization to significantly reduce the critical path and multiplication cost, still featuring the advantage of not increasing iteration count and needing extra correction.

MM Implementation and comparison. We conduct thorough comparisons with relevant designs to evaluate the profit of our optimizations, especially focusing on the plentiful 256-bit ASIC implementations. Table 8 lists the implementation results of modular multipliers based on different design strategies and algorithm types. Although the digit-serial and partially-pipelined strategies fluently lead to low-cost MM implementations, they are hard to meet the incredibly high-throughput requirement of NTT and MSM processors. To make an apple-to-apple comparison, we also synthesize our MM with TSMC 65nm process. Baseline 1.x denotes the modular multiplier design based on the prevalent high-radix Montgomery algorithm [KKAK96], while baseline 2.x extends this by applying constant multiplier to make the quotient determination cost-free. As expected by the theoretical analysis, whether in fully pipelined or digit-serial strategy, the area and latency of baseline 2.x series outperform those of baseline 1.x. As the most relevant and up-to-date fully-pipelined counterpart, [SNF+19] borrows the classical quotient-delayed algorithm but fails to figure out the CM. optimization and load balancing techniques. For a fairer comparison, we synthesize the design of [SNF+19] under the same process node, whose latency and area cost are approximately 1.8× and 1.3× higher than those of our work. [LWD+21] develops the small-radix digit-serial Montgomery multiplier, which achieves a high frequency at the expense of large number of cycles. As a result, the area efficiency measured by ATP is much lower than this work. [GL19][IIA19] also employ pipeline techniques to the full-word Montgomery multiplication, achieving low latency at the price of heavy area overhead. [MAQSS16] devises efficient partially-pipelined MMs for special NIST modulus by applying customized modular reduction method. [KLC16] introduces a scalable radix-4 Montgomery multiplier based on systolic array, which is capable of handling variable bit-width precision. However, this design incurs a considerable cycle count, making it better suited for resource-constrained IoT application.

5.2 Evaluations on the Overall Architecture

Area breakdown. Table 4 reports the area breakdown across main building components for the unifield accelerator. Thanks to the configurable and unifield design methodology, the total on-chip area of the accelerator only occupies 11.31 mm^2 area, which is about 4.3× lower than the sumed area of NTT and MSM kernels within PipeZK. As can be seen, the compact and configurable PE array supporting both NTT and MSM operations consumes 8.45 mm^2 area, accounting for around 74.7% of total area cost.

Table 4: The area breakdown across main sub-modules

Component	Area (mm^2)	Percentage (%)	Component	Area (mm^2)	Percentage (%)
PE Array	8.45	74.7	BankGroup4	0.52	4.6
⌊ 4×PEs (256-bit)	1.1	13	TF Engine	0.36	3.2
⌊ 12×PEs (384-bit)	7.35	87	⌊ 16×digit-serial MMs	0.25	69.4
BankGroup0-3	1.66	14.7	⌊ seed bank	0.11	30.6
Other Parts	0.32	2.8	Total Area	11.31	–

Table 8: The comparison about 256-bit Montgomery multipliers under ASIC synthesis

schemes	platforms	design strategy	algorithm type	Frequency (Hz)	Cycle	Latency	Area	ATP[c]
high-radix Montgomery modular multiplication from [KKAK96]								
Baseline 1.0	TSMC 28nm	fully pipelined	radix-16 Mont.	1G	16	16ns	0.4mm^2/793.3KGEs	6.4/12.7
Baseline 1.1		digit serial	radix-16 Mont.	1G	16	16ns	0.024mm^2/46.8KGEs	0.4/0.75
Baseline 1.2		fully pipelined	radix-32 Mont.	1G	8	8ns	0.29mm^2/581.5KGEs	2.3/4.7
Baseline 1.3		digit serial	radix-32 Mont.	0.9G	8	8ns	0.049mm^2/96.4KGEs	0.4/0.77
high-radix Montgomery modular multiplication from [KKAK96] + CM. opt.								
Baseline 2.0	TSMC 28nm	fully pipelined	radix-16 Mont.	1G	16	16ns	0.38mm^2/750.9KGEs	6.1/12
Baseline 2.1		digit serial	radix-16 Mont.	1G	16	16ns	0.022mm^2/43KGEs	0.35/0.69
Baseline 2.2		fully pipelined	radix-32 Mont.	1G	8	8ns	0.27mm^2/523KGEs	**2.16/4.2**[a]
Baseline 2.3		digit serial	radix-32 Mont.	1G	8	8ns	0.039mm^2/77KGEs	**0.32/0.62**[a]
[SNF+19]	TSMC 28nm	fully pipelined	radix-32 Mont.	1G	13	13ns	0.31mm^2/615KGEs	**4.1/8**[b]
[LWD+21]	TSMC 65nm	digit serial	radix-2 Mont.	1G	130	130ns	0.4mm^2	52/—
[LWD+21]	TSMC 65nm	digit serial	radix-4 Mont.	715M	66	92ns	31.7KGEs	—/2.9
[GL19]	SMIC 65nm	partially pipelined	full-word Mont.	498M	22	44ns	117.6KGEs	—/5.2
[IIA19]	TSMC 65nm	fully pipelined	full-word Mont.	238M	14	60ns	2.8mm^2	168/—
[MAQSS16]	GF 65nm	partially pipelined	NIST Reduction	244M	361	1.48μs	0.47mm^2/114KGEs	695/168
[KLC16]	TSMC 90nm	systolic array	radix-4 Mont.	794M	130	163ns	4mm^2	652/—
This work	TSMC 28nm	fully pipelined	mixed-radix Mont.	1.1G	8	7.3ns	0.24mm^2/484.1KGEs	**1.7/3.5**
		digit serial	radix-32 Mont.	1.2G	9	7.5ns	0.036mm^2/72.8KGEs	**0.27/0.55**
	TSMC 65nm[d]	fully pipelined	mixed-radix Mont.	425M	8	18.4ns	0.95mm^2/495KGEs	17.4/9.1
		digit serial	radix-32 Mont.	525M	9	17.1ns	0.16mm^2/83.3KGEs	2.7/1.4

NOTES: CM. opt. - constant multiplication optimization. [a]: deontes the best design among baseline implementations. [b]: denotes the most advanced and related work in terms of fully pipelined Montgomery multiplier. [c]: Area time product (ATP) is calculated as: x mm^2 $\times$ y ns or x KGEs $\times$ y ns / 1000. [d]: uses the same process node for fair comparison.

Table 9: The implementation and comparision about large-scale MSM architecture

scheme	platform	size	modulus bit-width	NTT Support	# PADD	window size	area (mm^2)	frequency (Hz)	cycle (M)	latency (ms)
[CPD+24] (GPU)	V100-SXM2 (12nm)	2^{24} 2^{22} 2^{20}	381-bit	Yes	80[a]	16 (4.5MB)	815	1.23G	1542.3 394.4 105.4	1233.87 (1.1×) 315.51 (1.1×) 84.28 (1.2×)
	RTX3090 (8nm)				82[a]		628.4	1.4G	1180.7 299.7 79.7	843.18 214.94 56.91
	RTX4090 (4nm)				128[a]		608	2.23G	1116.2 277.3 73.3	500.07 124.21 32.86
PipeZK [b] [ZWZ+21]	UMC 28nm	2^{20} 2^{18} 2^{16}	381-bit	No	2	4 (2.25KB)	33.72	300M	55.8 27.9 13.9	184 (2.5×) 92 (5.0×) 46 (10×)
PipeMSM [Xav22]	Alveo U55C	2^{20} 2^{18} 2^{16}	377-bit	No	1	12 (288KB)	–	125M	34.1 8.6 2.2	273 (3.7×) 68.8 (3.8×) 17.6 (3.8×)
CycloneMSM [ABC+22]	VU9P FPGA	2^{26} 2^{24} 2^{22}	377-bit	No	1	16 (4.5MB)	–	250M	1414 440.3 204.5	5656 (1.2×) 1761 (1.5×) 817.9 (2.8×)
HardCaml[b] [RDQY24]	VU9P FPGA	2^{26}	377-bit	No	1	13 (576KB)	–	278M	1419.4	4968 (1.1×)
This work[c]	**TSMC 28nm**	2^{26} 2^{24} 2^{22} 2^{20} 2^{18} 2^{16}	**381-bit**	**Yes**	**1**	**6/ 13 (4.5KB/ 288KB)**	**11.31/ 15.04**	**800M**	**3731.1/1737.2 932.8/435.7 233.2/110.2 58.3/28.9 14.6/8.6 3.7/3.5**	**4663.9/2171.5 1166.0/544.6 291.5/137.8 72.9/36.1 18.3/10.7 4.6/4.4**
					2	**6/ 13 (4.5KB/ 288KB)**	**22.62/ 30.08**	**800M**	**2072.8/965.1 518.2/242.0 129.6/61.3 32.4/16.1 8.1/4.8 2.0/1.9**	**2591.1/1206.4 647.8/302.5 161.9/76.6 40.5/20.1 10.1/6.0 2.6/2.4**

[a]: the number of streaming multiprocessors in GPU. [b]: only reports the cycle count of bucket accumulation phase. [c]: window size $c = 6$ is chosen as the default comparison object. the reported latency covers all three phases of MSM.

Table 5: The theoretical analysis and comparison about different NTT/INTT variants

schemes	sub-NTT	sub-INTT	BFU type	Unified BFU Overhead	Transpose Overhead	TF Storage
baseline	DIF-NR	DIT-RN	CT+GS	1 MM+2 MA+ 2 MS+4 MUX	Y	2n
[ZWZ+21]	DIF-NR	DIT-RN	CT+GS	1 MM+2 MA+ 2 MS+4 MUX	Y	2n
[WG23]	DIF-NR	No support	GS	1 MM+1 MA+ 1 MS	Y	n
Proteus [HMR23]	DIF-NR	DIF-RN	GS	1 MM+1 MA+ 1 MS+2 MUX	N	n/2
Supranational	DIF-NR	No support	GS	1 MM+1 MA+ 1 MS	N	n
HardCaml	DIF-NR	No support	GS	1 MM+1 MA+ 1 MS	Y	n
This work	**DIF-NR**	**DIF-RN**	**GS**	**1 MM+1 MA+ 1 MS+2 MUX**	**N**	**n/2**

NOTES: n denotes the length of vector. MM, MA, MS denote the modular multiplier, modular adder and modular subtracter, respectively.

Comparisons about the NTT algorithm. Table 5 presents the algorithms used by recent studies on ZKP-oriented NTT accelerator. [ZWZ+21] adopts the same type of sub-NTT as the baseline. Its unified BFU requires more modular additions and subtractions than our work. It also takes up more memory footprint for TFs and needs extra matrix transposition. [WG23] further extends the two-dimensional NTT form into higher dimensions, allowing for the flexible handling of vectors with different sizes. Unfortunately, this extension consumes proportionally more Hadamard products and transposition cost. [HMR23] conducts a thorough analysis on the bit-reversed issue and proposes a low-cost unified butterfly unit. However, it introduces extra registers into BFU for timing-synchronization issues inherent in SDF and MDC architectures. The teams Supranational and HardCaml, in response to the NTT parameters specified by Zprize 2022, also adopt the four-step NTT algorithm. Nonetheless, their approaches merely support the NTT mode and require extra cycle count for matrix transposition as well.

NTT implementation and comparison. Table 6 presents the implementation of NTT proposed by recent works considering different scales and platforms. The software platform employs an Intel Xeon Gold 5120 CPU with 14 physical cores running at 2.2 GHz, alongside 252 GB DDR4 memory, leveraging the bellman library [Cor22] to compute NTT at various scales. In contrast, our design achieves over 3000× acceleration on average. PipeZK utilizes the same process node and off-chip memory simulator as our work, but falls short of optimizing the large-width modular multiplier. The frequency of NTT kernel is about 2.7× lower than our work. As a result, our design achieves almost 12.1× acceleration and consumes 1.4× lower area cost compared to PipeZK (15.04 mm^2). SAM implements a scalable large-scale NTT design on FPGA platform. However, the adopted SDF architecture only achieves 50% hardware utilization and the modular multiplier is not fully pipelined, which leads to around 3.1× larger latency than our work. PROTEUS provides an automated design tool capable of generating SDF and MDC-type pipelined NTT hardware kernels under different parameters, but it takes no account of the latency of DRAM access. The top two contestants in Zprize 2022 devise large-scale NTT processors tailored for 64-bit Goldilocks modulus, both of which employ HBM to obtain high memory bandwidth. Thanks to the special modular reduction methods, these two works easily achieve high frequency and decent performance but only support NTT mode. The tiled DRAM layout in this work can still be applied to these two works to avoid transposition overhead, thereby offering the potential to further enhance overall performance.

MSM implementation and comparison. Similar to NTT, there are many designs

Table 6: The implementation and comparision about large-scale NTT/INTT architecture

schemes	platform	size	modulus (bits)	frequency (Hz)	latency (ms)	memory
CPU	Intel Xeon	2^{16}	256	2.2G	511.47	off-chip
		2^{20}			1244.68	
		2^{24}			19970.75	
PipeZK [ZWZ+21]	UMC 28nm	2^{16}	256	300M	0.281	DDR+FIFO
		2^{20}			11	
SAM [WG23]	XCU250	2^{16}	256	100M	1.24	DDR+FIFO
		2^{20}			12.61	
		2^{24}			183.56	
PROTEUS [HMR23]	XCU250	2^{16}	256	125M	1.05	on-chip
SAM [WG23]	XCU250	2^{16}	64	165M	0.38	DDR+FIFO
		2^{20}			2.84	
		2^{24}			34.12	
PROTEUS [HMR23]	XCU250	2^{16}	64	135M	0.44	on-chip
Supranational	Varium C1100	2^{18}	64	464M	2.47	HBM+BRAM
HardCaml	Varium C1100	2^{24}	64	—	168	HBM+BRAM
This work	TSMC 28nm	2^{16}	256	800M	0.047	DDR+SRAM
		2^{20}			0.91	

improving the performance of MSM with optimizations on GPU, ASIC and FPGA, as listed in Table 9. For the GPU-based MSM implementations, recent works [LWY+23][CPD+24] consecutively propose new parallel Pippenger algorithms to achieve nearly perfect linear speedup and obtain load-balanced computation pattern. Compared to the 2^{20}-scale MSM computation on the modern GPU V100 platform, our hardware accelerator still delivers a speedup of 1.2×. It is not easy to do an apple-to-apple comparison to the MSM implementation on FPGA, since FPGA and ASIC platforms each have their own advantages. For instance, FPGAs possess abundant memory and parallel computing resources, but their implementation frequency is relatively lower due to the complex routing. In contrast, the area cost of ASIC implementation is sensitive to the memory capacity, which is the reason why we choose a relatively small value $c = 6$ and leverage limited on-chip banks. PipeMSM profiles the variation of cycle count with the window size c, and then selects a value $c = 12$, which is much larger than our chosen case. CycloneMSM carefully develops a scheduler to improve resource utilization, and adopts the mixed addition formula in extended Twisted Edwards coordinates to trade computation with memory. It even chooses a larger window size $c = 16$, capable of handling 2^{26}-scale MSM. HardCaml features a split CPU-FPGA architecture that amortizes the bucket/group aggregation phase to the software platform. The window size is set as $c = 13$ in HardCaml. On average, the speed of our work still outperforms the above FPGA designs by 3.7×, 1.8× and 1.1×, which can be even promoted significantly if we further increase the memory resource and number of PADD. Compared to the most related and advanced ASIC implementation, our work offers an average speedup of 5.8× while consumes 3.1× lower area cost. This superiority in performance can be attributed to the highly optimized modular multiplier, configurable PE array and dedicated scheduling mechanism.

Sensitivity study on the window size. To further enhance the performance improvement of the proposed MSM hardware accelerator against software implementations on the high-end powerful RTX3090/4090 GPU [CPD+24], Table 9 also manifests the latency and area of MSM accelerator when setting the window size as $c = 13$ and using two parallel kernels along with on-chip memory (#PADD = 2). As illustrated in [ZWZ+21, LWY+23], the $h = \lceil \frac{\lambda}{c} \rceil$ sub-tasks across different windows are independent from each other and can

be computed in parallel. At this time, compared to the software implementation on the most powerful RTX4090 GPU, the speedup ratio of our accelerator is raised up to about 1.66×/1.62×/1.63× when computing $2^{18}/2^{20}/2^{22}$-scale MSM, respectively. Notably, the area consumption of our 28nm accelerator is 20.3× lower than that of RTX4090 GPU even fabricated with 4nm process, directly leading to much higher area and power efficiency. Additionally, we explore how the MSM latency of different sizes varies with the window size c and the segmented degree, giving some design rationale for speed-priorized MSM accelerators. The segmented degree refers to the number of partitioned segments during the bucket aggregation phase using the method proposed by [Xav22]. Taking two typical sizes as the case study, Figure 14 profiles how the latency scales with the window size and how the segmented degree influences the curved shape, from which we make some crucial observations as blow. Evidently, increasing the window size to some degree helps reduce the overall MSM latency, and the turning point depends on the vector size and segmented degree. For the 256K-point MSM, the turning point of window size happens to be around 12-14, and enlarging the segmented degree also shifts the turning point to the right. For the 1M-point MSM, the turning point becomes much larger, and the segmented degree has a smaller impact on the curved shape. Thus, selecting both a suitable window size and segmented degree is of significance to seek sound balance between area and latency.

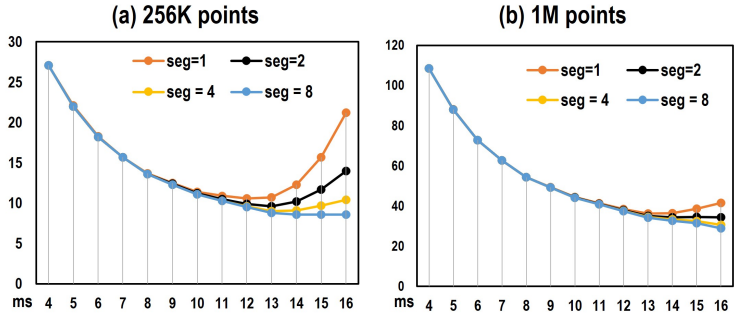


Figure 14: The variation of MSM latency with window size and segmented degree.

Discussion on the constant-time implementation. In the security-critical scenario, a constant-time implementation is necessary to mitigate risks of timing attacks, which motivates us to make an analysis from the underlying arithmetic unit to the high-level architecture. The conditional assignment shown in the line 19-20 of Algorithm 3 may lead to the variation of cycle count under software platforms. However, the proposed hardware implementation for the Montgomery multiplication invariably consumes 9 cycles no matter whether the condition meets or not. Thus, the underlying Montgomery multiplier is a constant-time implementation. For our NTT hardware design, the overall cycle count depends on the vector length N as shown in Section 4.2, because the on-chip butterfly computations and hadmard products are assumed to be the bottleneck for every batch processing, instead of the off-chip memory access. As mentioned in Section 2.1, the vector length is determined by the number of gates transformed form the circuit program and is a publicly known parameter. Therefore, the variation of NTT latency will not leak any information about the private witness vector.

As for the MSM computation, the overall latency is associated with the vector length n, the bit-width of scalar λ, the window size c and the value distribution of scalar vector $\{s_i\}_{i=[0,n-1]}$, since the Pippenger algorithm is computed as the formula: $O = \sum_{i=0}^{n-1} s_i \cdot P_i = \sum_{j=0}^{h-1} 2^{jc} \sum_{i=0}^{n-1} s_{i,j} \cdot P_i$, where $h = \lceil \frac{\lambda}{c} \rceil$. In the zk-SNARK protocol, the set of scalar $\{s_i\}_{i\in[0,n-1]}$ will be divided into two parts, w.l.o.g., with one part being the public inputs $\{s_i\}_{i\in[0,k-1]}$ and another part being the private witness $\{s_i\}_{i\in[k,n-1]}$. [ZHY+24] has already made a detailed analysis to show that the publicly-known parameters

n, λ, c will not leak any information about the private witness $\{s_i\}_{i\in[k,n-1]}$, which can be obviously confirmed from the workflow of zk-SNARK in Section 2.1 as well. [ZHY+24] also mentions that its MSM software program on GPU is independent of the distribution of scalar vector (secret inputs), since the available storage capacity is enough to assign independent bucket space to each thread. However, the MSM latency of [CPD+24] is related with the distribution of scalar vector, because a much larger window size is chosen, which requires multiple threads to share bucket space. As a result, the resolution of write conflicts may be influenced by the distribution of secret inputs, which motivates them to propose another constant-time version of MSM. Unlike the software program, the cycle count of our MSM hardware accelerator obviously depends on the distribution of scalar vector, since the load-aware arbitration mechanism inherently influences the overall cycle count. In practice, the random scalar vector is most evenly distributed, and the gap count between two extreme cases is small. Figure 15 presents the variation of cycle count for 1K-point MSM tested with 100 random scalar vectors. The cycle difference between the worst and best case is 56, taking up only 4.4% of the average cycle count. Thus, we would like to remark that the constant-time patch can be trivially implemented by padding the cycle of all cases to the worst case, and the overall performance degradation is still minor.

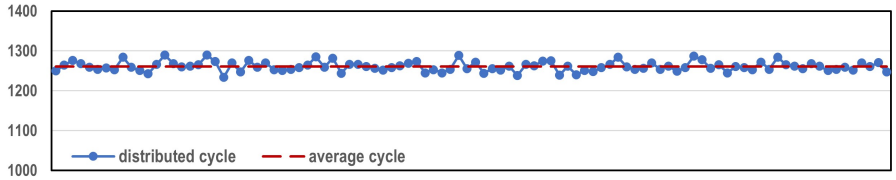


Figure 15: The variation of 1K-point MSM cycle with 100 random scalar vectors.

6 Conclusion

Despite the advancements in ZKP schemes, NTT and MSM still remain as the bottleneck, consuming a significant portion of the execution time. The recent hardware optimizations to accelerate ZKP are promising, but still use two separated data-paths for NTT and MSM, incurring tremendous area overhead. Our work introduces a unified and high-performance accelerator for both NTT and MSM, which employs the versatile scheduling mechanism, novel configurable PE array, meticulous memory partitioning and so on. We make a variety of optimizations from the modular arithmetic level to high-level NTT and MSM architecture. For the fully-pipelined modular multiplier design, two modularized, simple yet effective techniques are proposed to balance the pipelined workload and reduce the length of critical path. A proof-of-concept implementation of NTT and MSM achieves impressive improvement under TSMC 28nm synthesis. Compared to similar advanced works about modular multipliers, LBFP MM achieves a 1.8× improvement of latency and saves 1.3× area cost. The proposed accelerator demonstrates notable NTT and MSM speeds that are about 12.1× and 5.8× faster than those of the state-of-the-art ASIC designs, while still consuming 4.3× lower overall area cost. This research contributes to the ongoing efforts to enhance the practicality and efficiency of ZKP schemes, manifesting a promising stride towards making the proof-generation process more efficient and viable for real-world applications. Future work would discuss profiling the accelerator from ASIC to FPGA and adopting some optimization techniques specific to memory-rich FPGAs, such as width-oriented DSP utilization [LP21], custom LUT-based compressors [BRM19], and computation-memory trade-off. It is also interesting to add programmability to the accelerator for supporting various complete ZKP schemes. We will also investigate some side-channel protection techniques for accelerators deployed in security-critical scenarios.

Acknowledgements

This work is supported in part by the National Key R&D Program of China (Grant No.2023YFB4403500), and in part by the National Natural Science Foundation of China(Grant No. 62274102). We thank the editors and reviewers for their thoughtful comments.

References

[ABC+22] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. FPGA acceleration of multi-scalar multiplication: Cyclonemsm. *IACR Cryptol. ePrint Arch.*, page 1396, 2022.

[AFH16] Berkin Akin, Franz Franchetti, and James C. Hoe. FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation. *J. Signal Process. Syst.*, 85(1):67–82, 2016.

[AV20] hujw77 Alex Vlasov. Eip-2539: Bls12-377 curve operations [draft], February 2020.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 701–732. Springer, 2019.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. Faster batch forgery identification. In Steven D. Galbraith and Mridul Nandi, editors, *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings*, volume 7668 of *Lecture Notes in Computer Science*, pages 454–473. Springer, 2012.

[BFV19] Alex Biryukov, Daniel Feher, and Giuseppe Vitto. Privacy aspects and subliminal channels in zcash. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1795–1811. ACM, 2019.

[BL] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database.

[BL95] Wieb Bosma and Hendrik W. Lenstra. Complete systems of two addition laws for elliptic curves. *Journal of Number Theory*, 53:229–240, 1995.

[BRM19] Debapriya Basu Roy and Debdeep Mukhopadhyay. High-speed implementation of ecc scalar multiplication in gf(p) for generic montgomery curves. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(7):1587–1600, 2019.

[BS91] Wayne P. Burleson and Louis L. Scharf. A VLSI design methodology for distributed arithmetic. *J. VLSI Signal Process.*, 2(4):235–252, 1991.

[CA07] S. Chandrasekaran and A. Amira. Novel sparse obc based distributed arithmetic architecture for matrix transforms. In *2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3207–3210, 2007.

[CDF+11] Ray C. C. Cheung, Sylvain Duquesne, Junfeng Fan, Nicolas Guillermin, Ingrid Verbauwhede, and Gavin Xiaoxu Yao. FPGA implementation of pairings using residue number system and lazy reduction. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 421–441. Springer, 2011.

[Cor22] ZKCrypto Corp. bellman: a crate for building zk-snark circuits, 2022.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 769–793. Springer, 2020.

[CPD+24] Yutian Chen, Cong Peng, Yu Dai, Min Luo, and Debiao He. Load-balanced parallel implementation on GPUs for multi-scalar multiplication algorithm. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):522–544, 2024.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CYY+22] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. CFNTT: scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):94–126, 2022.

[dar19] darpa. Generating zero-knowledge proofs for defense capabilities, 2019.

[DL18] Jinnan Ding and Shuguo Li. A modular multiplier implemented with truncated multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(11):1713–1717, 2018.

[GL19] Zhen Gu and Shuguo Li. A division-free toom–cook multiplication-based montgomery modular multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 66(8):1401–1405, 2019.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

[HCG05] Yajuan He, Chip-Hong Chang, and Jiangmin Gu. An area efficient 64-bit square root carry-select adder for low power applications. In *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 4082–4085 Vol. 4, 2005.

[HMR23] Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. Proteus: A tool to generate pipelined number theoretic transform architectures for fhe and zkp applications. *IACR Cryptol. ePrint Arch.*, 2023:267, 2023.

[HWCD08] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 326–343, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[IIA19] Makoto Ikeda, Tadayuki Ichihashi, and Hiromitsu Awano. 33us, 94uj optimal ate pairing engine on bn curve over 254b prime field in 65nm cmos fdsoi. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 263–266, 2019.

[KKAK96] C. Kaya Koc, T. Acar, and B.S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.

[KLC16] Shiann-Rong Kuang, Chih-Yuan Liang, and Chun-Chi Chen. An efficient radix-4 scalable architecture for montgomery modular multiplication. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(6):568–572, 2016.

[Kun85] S. Y. Kung. Vlsi array processors. *IEEE ASSP Magazine*, 2:4–22, 1985.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zksnarks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):358–380, 2023.

[Lon23] Patrick Longa. Efficient algorithms for large prime characteristic fields and their application to bilinear pairings. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):445–472, 2023.

[LP21] Martin Langhammer and Bogdan Pasca. Efficient FPGA modular multiplication implementation. In Lesley Shannon and Michael Adler, editors, *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, pages 217–223. ACM, 2021.

[LTB+23] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, and Onur Mutlu. Ramulator 2.0: A modern, modular, and extensible dram simulator, 2023.

[LWD+21] Bing Li, Jinlei Wang, Guocheng Ding, Haisheng Fu, Bingjie Lei, Haitao Yang, Jiangang Bi, and Shaochong Lei. A high-performance and low-cost montgomery modular multiplication based on redundant binary representation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(7):2660–2664, 2021.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuzk: Accelerating zero-knowledge proof with A faster parallel multi-scalar multiplication algorithm on GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):194–220, 2023.

[LZD+24] Changxu Liu, Hao Zhou, Patrick Dai, Li Shang, and Fan Yang. Priormsm: An efficient acceleration architecture for multi-scalar multiplication. *ACM Trans. Des. Autom. Electron. Syst.*, jul 2024. Just Accepted.

[MAK+23] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded hardware accelerator for computing on encrypted data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):463–500, 2023.

[MAQSS16] Hamad Marzouqi, Mahmoud Al-Qutayri, Khaled Salah, and Hani Saleh. A 65nm asic based 256 nist prime field ecc processor. In *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2016.

[MK22] Ahmet Can Mert, Emre Karabulut, Erdinç Öztürk, Erkay Savaş, and Aydin Aysu. An extensive study of flexible design methods for the number theoretic transform. *IEEE Transactions on Computers*, 71(11):2829–2843, 2022.

[MLPJ13] Yuan Ma, Zongbin Liu, Wuqiong Pan, and Jiwu Jing. A high-speed elliptic curve cryptographic processor for generic curves over gf(p). In Tanja Lange, Kristin E. Lauter, and Petr Lisonek, editors, *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *Lecture Notes in Computer Science*, pages 421–437. Springer, 2013.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[MR18] Debdeep Mukhopadhyay and Debapriya Basu Roy. Revisiting FPGA implementation of montgomery multiplier in redundant number system for efficient ECC application in gf(p). In *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*, pages 323–326. IEEE Computer Society, 2018.

[MXS+23] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. GZKP: A GPU accelerated zero-knowledge proof system. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 340–353. ACM, 2023.

[Oru95] Holger Orup. Simplifying quotient determination in high-radix modular multiplication. In *12th Symposium on Computer Arithmetic (ARITH-12 '95), July 19-21, 1995, Bath, England, UK*, page 193. IEEE Computer Society, 1995.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 258–263, 1976.

[PR18] Somnath Panja and Bimal Kumar Roy. A secure end-to-end verifiable e-voting system using zero knowledge based blockchain. *IACR Cryptol. ePrint Arch.*, page 466, 2018.

[Pru22] Alex Pruden. Announcing the inaugural zprize competition results, 2022.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 403–428, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[RDQY24] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, page 33–39, New York, NY, USA, 2024. Association for Computing Machinery.

[Sch96] J.C. Schatzman. Index mappings for the fast fourier transform. *IEEE Transactions on Signal Processing*, 44(3):717–719, 1996.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, pages 238–252. ACM, 2021.

[SNF+19] Junichi Sakamoto, Yusuke Nagahama, Daisuke Fujimoto, Yota Okuaki, and Tsutomu Matsumoto. Low-latency pairing processor architecture using fully-unrolled quotient pipelining montgomery multiplier. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6, 2019.

[TAL93] Richard Tolimieri, Myoung An, and Chao Lu. *Multidimensional Tensor Product and FFT*, pages 29–43. Springer US, New York, NY, 1993.

[Tea22] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri, 2022.

[WB19] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):154–179, 2019.

[WG23] Cheng Wang and Mingyu Gao. Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[Xav22] Charles F. Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. *IACR Cryptol. ePrint Arch.*, page 999, 2022.

[XHY+20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 67-I(8):2672–2684, 2020.

[XZL+23] Zhibo Xing, Zijian Zhang, Jiamou Liu, Ziang Zhang, Meng Li, Liehuang Zhu, and Giovanni Russello. Zero-knowledge proof meets machine learning in verifiability: A survey, 2023.

[YZF+23] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. Poseidon: Practical homomorphic encryption accelerator. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 870–881. IEEE, 2023.

[ZCP23] Bo Zhang, Zeming Cheng, and Massoud Pedram. An iterative montgomery modular multiplication algorithm with low area-time product. *IEEE Trans. Computers*, 72(1):236–249, 2023.

[ZGK+17] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 863–880. IEEE Computer Society, 2017.

[ZHLH23] Baoze Zhao, Wenjin Huang, Tianrui Li, and Yihua Huang. Bstmsm: A high-performance fpga-based multi-scalar multiplication hardware accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 35–43, 2023.

[ZHY+24] Xudong Zhu, Haoqi He, Zhengbang Yang, Yi Deng, Lutan Zhao, and Rui Hou. Elastic MSM: A fast, elastic and modular preprocessing technique for multi-scalar multiplication algorithm on gpus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):258–284, 2024.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*, pages 416–428. IEEE, 2021.

Appendix

Table 7: The constant factors under different radices for BLS12-381 curve.

Radices	$\mathbb{F}_M$(256-bit)	$\mathbb{F}_Q$(384-bit)
$R_1 = 2^{32}$	0xffffffff	0xfffcfffd
$R_2 = 2^{32}$	0xffffffff	0xfffcfffd
$R_3 = 2^{48}$	0xfffeffffffff	0xfffcfffcfffd
$R_4 = 2^{64}$	0xfffffffeffffffff	0x89f3fffcfffcfffd
$R_5 = 2^{80}$	0xfffe5bfdfffffffeffffffff	0x13e889f3fffcfffcfffd
$R_6 = 2^{128}$		0x286adb92d9d113e889f3fffcfffcfffd

A Details for MSM Mode

Figure 16 presents the concrete dataflow of PADD unit under the complete point addition formula [RCB16][BL95], where the affine coordinate is replaced by the homogeneous projective coordinates: $(x, y) \rightarrow (X/Z, Y/Z)$. After serious investigations into [BL][HWCD08], it is believed that another appealing choice is the point addition formula under extended Twisted Edwards coordinates, which also obtains judicious trade-off between computation and memory. As introduced in Section 3.1 of [HWCD08], it is strongly unifield and complete, necessitating a total of 9 MMs, 2 CMs and 9 MAs. Although the configurable and unified design methodology can be readily adapted to the new choice of formula, only one PADD unit is allowed by reusing the given #16 PEs as well. Thus, the improvement in terms of speed would be slight from the holistic and practical view. Figure 17 portrays the timing diagram for the 16-point bucket accumulation under one slice of the scalar vector, where the arbitration cases are explictly labelled at every cycle.

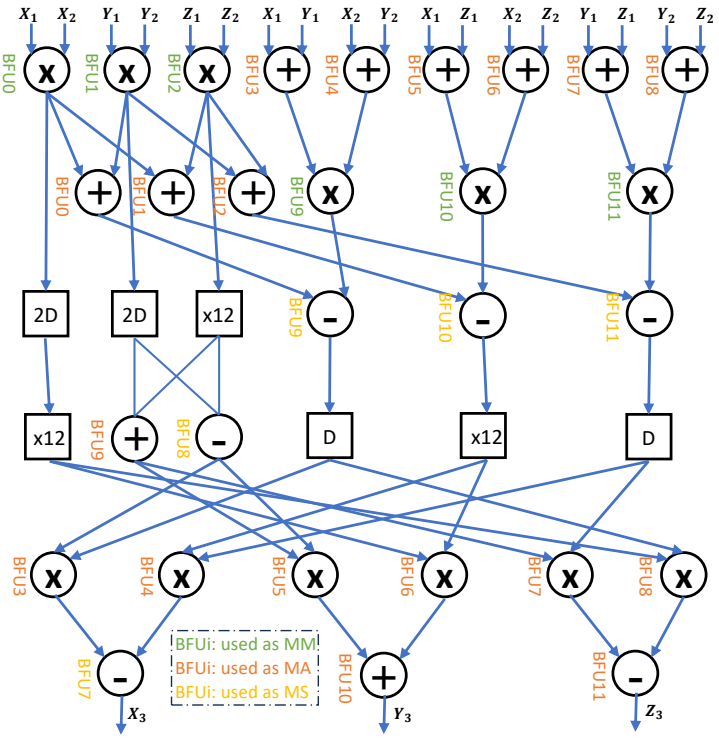
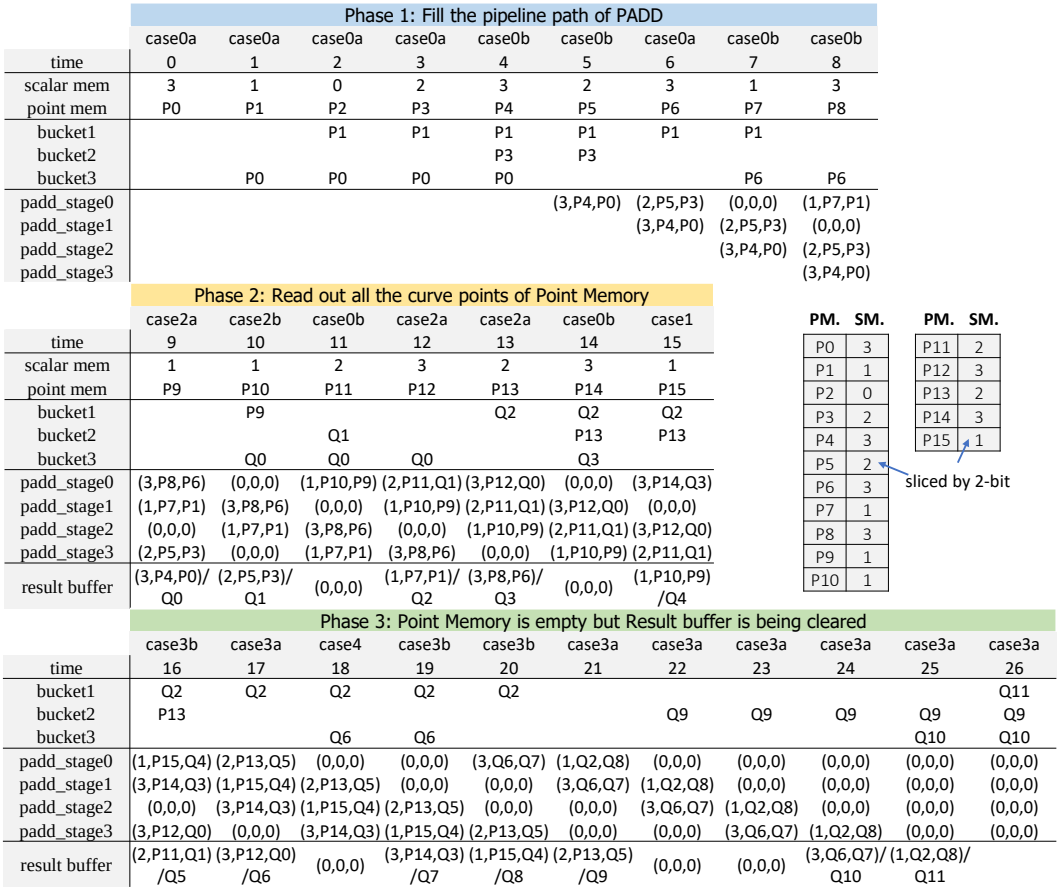


Figure 16: The dataflow of PADD and interconnections among 12 PEs.



Phase 1: Fill the pipeline path of PADD									
	case0a	case0a	case0a	case0a	case0b	case0b	case0a	case0b	case0b
time	0	1	2	3	4	5	6	7	8
scalar mem	3	1	0	2	3	2	3	1	3
point mem	P0	P1	P2	P3	P4	P5	P6	P7	P8
bucket1			P1	P1	P1	P1	P1	P1	
bucket2					P3	P3			
bucket3		P0	P0	P0	P0			P6	P6
padd_stage0						(3,P4,P0)	(2,P5,P3)	(0,0,0)	(1,P7,P1)
padd_stage1							(3,P4,P0)	(2,P5,P3)	(0,0,0)
padd_stage2								(3,P4,P0)	(2,P5,P3)
padd_stage3									(3,P4,P0)

Phase 2: Read out all the curve points of Point Memory							
	case2a	case2b	case0b	case2a	case2a	case0b	case1
time	9	10	11	12	13	14	15
scalar mem	1	1	2	3	2	3	1
point mem	P9	P10	P11	P12	P13	P14	P15
bucket1		P9			Q2	Q2	Q2
bucket2			Q1			P13	P13
bucket3		Q0	Q0	Q0		Q3	
padd_stage0	(3,P8,P6)	(0,0,0)	(1,P10,P9)	(2,P11,Q1)	(3,P12,Q0)	(0,0,0)	(3,P14,Q3)
padd_stage1	(1,P7,P1)	(3,P8,P6)	(0,0,0)	(1,P10,P9)	(2,P11,Q1)	(3,P12,Q0)	(0,0,0)
padd_stage2	(0,0,0)	(1,P7,P1)	(3,P8,P6)	(0,0,0)	(1,P10,P9)	(2,P11,Q1)	(3,P12,Q0)
padd_stage3	(2,P5,P3)	(0,0,0)	(1,P7,P1)	(3,P8,P6)	(0,0,0)	(1,P10,P9)	(2,P11,Q1)
result buffer	(3,P4,P0)/ Q0	(2,P5,P3)/ Q1	(0,0,0)	(1,P7,P1)/ Q2	(3,P8,P6)/ Q3	(0,0,0)	(1,P10,P9) /Q4

PM.	SM.
P0	3
P1	1
P2	0
P3	2
P4	3
P5	2
P6	3
P7	1
P8	3
P9	1
P10	1

PM.	SM.
P11	2
P12	3
P13	2
P14	3
P15	1

sliced by 2-bit

Phase 3: Point Memory is empty but Result buffer is being cleared											
	case3b	case3a	case4	case3b	case3b	case3a	case3a	case3a	case3a	case3a	case3a
time	16	17	18	19	20	21	22	23	24	25	26
bucket1	Q2	Q2	Q2	Q2	Q2						Q11
bucket2	P13						Q9	Q9	Q9	Q9	Q9
bucket3			Q6	Q6						Q10	Q10
padd_stage0	(1,P15,Q4)	(2,P13,Q5)	(0,0,0)	(0,0,0)	(3,Q6,Q7)	(1,Q2,Q8)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
padd_stage1	(3,P14,Q3)	(1,P15,Q4)	(2,P13,Q5)	(0,0,0)	(0,0,0)	(3,Q6,Q7)	(1,Q2,Q8)	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)
padd_stage2	(0,0,0)	(3,P14,Q3)	(1,P15,Q4)	(2,P13,Q5)	(0,0,0)	(0,0,0)	(3,Q6,Q7)	(1,Q2,Q8)	(0,0,0)	(0,0,0)	(0,0,0)
padd_stage3	(3,P12,Q0)	(0,0,0)	(3,P14,Q3)	(1,P15,Q4)	(2,P13,Q5)	(0,0,0)	(0,0,0)	(3,Q6,Q7)	(1,Q2,Q8)	(0,0,0)	(0,0,0)
result buffer	(2,P11,Q1) /Q5	(3,P12,Q0) /Q6	(0,0,0)	(3,P14,Q3) /Q7	(1,P15,Q4) /Q8	(2,P13,Q5) /Q9	(0,0,0)	(0,0,0)	(3,Q6,Q7)/ Q10	(1,Q2,Q8)/ Q11	

Figure 17: An example of timing diagram for 16-point bucket accumulation.

B Off-chip Memory Access Scheme

We adopt the four-step NTT algorithm to decompose lengthy one-dimension vectors into appropriately sized two-dimensional matrices, facilitating the parallel and independent processing of each row and column using sub-NTTs. However, if we simply store the matrix in DRAM row by row (or column by column), it may incur large strides when accessing the columns (or rows) of the matrix, leading to frequent row buffer miss. To mitigate this issue and avoid additional overhead of matrix transposition, we adopt a tiled memory layout approach to store the matrix [AFH16], as illustrated in Figure 18. First, we reorganize the original vector data into a logical $N_R \times N_C$ matrix, which is further partitioned into data blocks of $N_x \times N_y$ dimension. The second step entails mapping each data block to every row in DRAM, ensuring that accessing the data within the block will not result in row buffer misses. Finally, Figure 18 illustrates the rearranged DRAM layout along with the address sequences when performing row-wise and column-wise sub-NTT, respectively. Here, we try to organize the data in a square matrix form ($N_x = N_y$), which aims to fully utilize the on-chip memory during either row-wise or column-wise sub-NTT. The black dashed boxes in Figure 18 denote the maximal amount of data that can be accommodated into on-chip memory banks, specifically equal to N-row or M-column data blocks, which is also referred to as one *batch* of data transfer from off-chip to on-chip memory.

As can be observed from Figure 18, accessing data in blocks ensures that both row-wise and column-wise access are performed with almost contiguous address, which averts the

impact of stride access and minimizes the frequency of row buffer miss. To implement the tiled DRAM data layout, we need to devise a two-layer address mapping logic interface. The first-layer logic interface maps the row-wise addresses of matrix into the tiled-wise addresses, which are comprised of inter-block and intra-block row/column IDs. The second-layer logic interface then converts the obtained tiled addresses into physical addresses for the DRAM, including the channel, rank, bank group, bank, row, and column fields. This DRAM layout can be readily initialized during the data transfer from CPU to DDR, which requires no extra execution time and other resource cost.

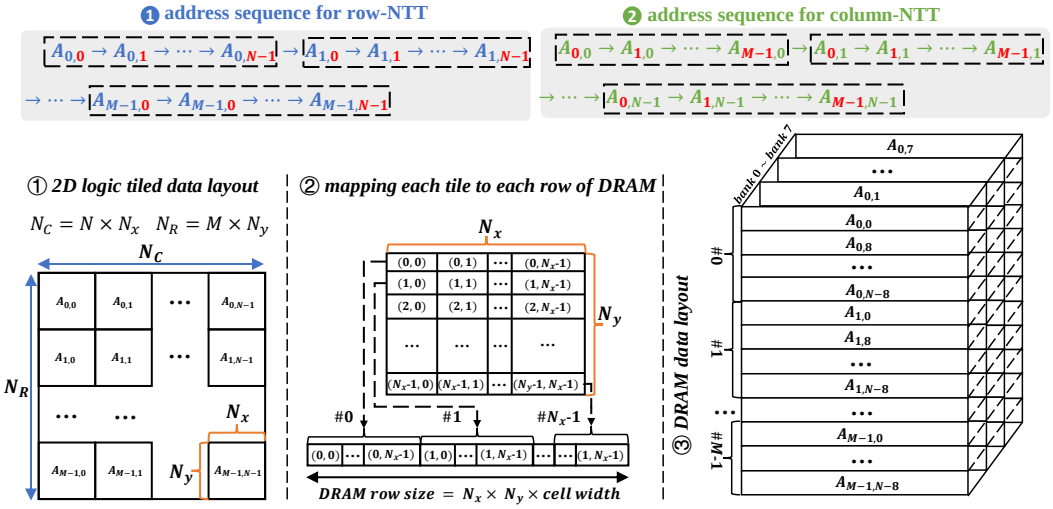


Figure 18: The mapping process from tiled matrix to DRAM layout.

C Index-mapping based Four-step NTT

First, the basic form is written as:

$$A_i = \sum_{j=0}^{N-1} a_j \cdot \omega_N^{ij} \bmod M, i = 0, 1, ..., N-1. \tag{8}$$

If we map the one-dimension vector of length N into the $N_1 \times N_2$ two-dimensional matrix, i.e., $n = N_2 \cdot n_1 + n_2$ for $n = 0, 1, ..., N-1; n_1 = 0, 1, ..., N_1 - 1; n_2 = 0, 1, ..., N_2 - 1$, the indices i and j can be expressed as a bivariate function:

$$j = N_2 \cdot j_1 + j_2 \bmod N, \quad i = i_1 + N_1 \cdot i_2 \bmod N, \quad N = N_1 \cdot N_2 \tag{9}$$

Here, $j_1, i_1 \in [0, N_1 - 1]$ and $j_2, i_2 \in [0, N_2 - 1]$. Thus, we obtain the following expression:

$$A_{i_1+N_1 \cdot i_2} = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} a_{N_2 \cdot j_1 + j_2} \cdot \omega_N^{ij} \bmod M \tag{10}$$

Defining the two-dimention array $\hat{A}$ and $\hat{a}$ gives that:

$$\hat{A}[i_1, i_2] = \sum_{j_1=0}^{N_1-1} \sum_{j_2=0}^{N_2-1} \hat{a}[j_1, j_2] \cdot \omega_N^{ij} \bmod M \tag{11}$$

with $\omega_N^{ij} = \omega_N^{i_1 \cdot j_2 \bmod N} \cdot \omega_{N_1}^{i_1 \cdot j_1 \bmod N} \cdot \omega_{N_2}^{i_2 \cdot j_2 \bmod N}$. Building upon the above derivation, we can further apply the decomposition technique to vectors of length N_1 and N_2, respectively. Finally, we obtain the basic form of four-step NTT:

$$\hat{A}[i_1, i_2] = \sum_{j_1=0}^{N_1-1} \left\{ \left[\sum_{j_2=0}^{N_2-1} \hat{a}[j_1, j_2] \cdot \omega_{N_2}^{i_2 \cdot j_2} \right] \cdot \omega_N^{i_1 \cdot j_2} \right\} \cdot \omega_{N_1}^{i_1 \cdot j_1} \bmod M \tag{12}$$

Algorithm 4 Optimized four-step NTT/INTT algorithm

Input: $a(x)$ is the polynomial of degree $N-1$ with coefficients forming the vector $\mathbf{a} = (a_0, a_1, ..., a_{N-1})$. ω_N is the N-th primitive root of unity over $\mathbb{F}_M$. $N = N_1 \times N_2$.
Output: $A(x) =$ Four-step-NTT$(a(x))$, $a(x) =$ Four-step-INTT$(A(x))$.

Rearrange the vector $\mathbf{a}$ *into column-wise matrix:*
$\hat{a}_{N_1 \times N_2} = matrix(\mathbf{a})$ ▷ the element at row i, column j is $\hat{a}[i,j] \in \hat{a}_{N_1 \times N_2}$.
Perform four-step NTT:
STEP 1: Perform sub-NTT for each row:
for $i = 0$ to $N_1 - 1$ **do**
    $\hat{a}[i,:] =$ DIF-NR-NTT$(\hat{a}[i,:])$ ▷ natural order input and bit-reversed order output.
end for
STEP 2: Perform inner product with twiddle factor:
for $i = 0$ to $N_1 - 1$ **do**
    for $j = 0$ to $N_2 - 1$ **do**
        $\hat{a}[i,j] = \hat{a}[i,j] \cdot \omega_N^{i \cdot \text{bit-reversed}(j)} \bmod M$
    end for
end for
STEP 3: Perform transposition: ▷ this operation could be avoided by using tiled layout.
$\hat{a}_{N_2 \times N_1} = \hat{a}^T_{N_1 \times N_2}$
STEP 4: Perform sub-NTT for each column:
for $i = 0$ to $N_2 - 1$ **do**
    $\hat{a}[i,:] =$ DIF-NR-NTT$(\hat{a}[i,:])$ ▷ natural order input and bit-reversed order output.
end for
$A(x) = vector(\hat{a}_{N_2 \times N_1})$
Perform four-step INTT:
STEP 1: Perform sub-INTT for each column:
for $j = 0$ to $N_2 - 1$ **do**
    $\hat{a}[i,:] =$ DIF-RN-INTT$(\hat{a}[i,:])$ ▷ bit-reversed order input and natural order output.
end for
STEP 2: Perform inner product with twiddle factor:
for $i = 0$ to $N_2 - 1$ **do**
    for $j = 0$ to $N_1 - 1$ **do**
        $\hat{a}[i,j] = -\hat{a}[i,j] \cdot \omega_N^{N/2 - \text{bit-reversed}(i) \cdot j} \bmod M$ ▷ reusing the twiddle factor of NTT.
    end for
end for
STEP 3: Perform transposition: ▷ this operation could be avoided by using tiled layout.
$\hat{a}_{N_1 \times N_2} = \hat{a}^T_{N_2 \times N_1}$
STEP 4: Perform sub-INTT for each row:
for $i = 0$ to $N_1 - 1$ **do**
    $\hat{a}[i,:] =$ DIF-RN-INTT$(\hat{a}[i,:])$ ▷ bit-reversed order input and natural order output.
end for
$a(x) = vector(\hat{a}_{N_1 \times N_2})$
return $A(x), a(x)$

Algorithm 5 DIF-NR radix-2 NTT algorithm

Input: $a(x)$ is the polynomial of degree n-1 with coefficients forming the vector $\mathbf{a} = (a_0, a_1, ..., a_{n-1})$. ω_n is the n-th primitive root of unity over $\mathbb{F}_M$.
Output: $A(x) = \text{DIF-NR-NTT}(a(x))$.

Precompute the twiddle factors:
for $i = 0$ to $n/2 - 1$ **do**
    $\omega\_\text{ROM}[i] = \omega_n^i$
end for
Perform the NTT operation:
for $p = log_2 n - 1$ to 0 **do**
    $J = 2^p$
    $S = n/(2 \cdot J)$
    for $k = 0$ to $S - 1$ **do**
        for $j = 0$ to $J - 1$ **do**
            $\omega = \omega\_\text{ROM}[j \cdot S]$ ▷ read out the twiddle factor.
            $u = a_{2 \cdot k \cdot J + j}$
            $v = a_{2 \cdot k \cdot J + j + J}$
            $a_{2 \cdot k \cdot J + j} = u + v \bmod M$
            $a_{2 \cdot k \cdot J + j + J} = (u - v) \cdot \omega \bmod M$ ▷ Gentleman-Sande BFU.
        end for
    end for
end for
return $A(x) = \mathbf{a}$ ▷ bit-reversed order output.

Algorithm 6 DIF-RN radix-2 INTT algorithm

Input: $a(x)$ is the polynomial of degree n-1 with coefficients forming the vector $\mathbf{a} = (a_0, a_1, ..., a_{n-1})$. ω_n is the n-th primitive root of unity over $\mathbb{F}_M$.
Output: $A(x) = \text{DIF-RN-INTT}(a(x))$.

for $p = 0$ to $log_2 n - 1$ **do**
    $J = 2^p$
    $S = n/(2 \cdot J)$
    for $k = 0$ to $S - 1$ **do**
        $\omega = -\omega\_\text{ROM}[n/2 - \text{bit-reversed}(k)] \bmod M$ ▷ reuse the twiddle factor of NTT.
        for $j = 0$ to $J - 1$ **do**
            $u = a_{2 \cdot k \cdot J + j}$
            $v = a_{2 \cdot k \cdot J + j + J}$
            $a_{2 \cdot k \cdot J + j} = (u + v)/2 \bmod M$
            $a_{2 \cdot k \cdot J + j + J} = [(u - v) \cdot \omega]/2 \bmod M$ ▷ Gentleman Sande IBFU.
        end for
    end for
end for
return $A(x) = \mathbf{a}$ ▷ natural order output.