TPU as Cryptographic Accelerator

Rabimba Karanjai
University Of Houston
USA
rkaranja@cougarnet.uh.edu

Sangwon Shin
Korea University
South Korea
husask11@korea.ac.kr

Wujie Xiong
Kent State University
USA
xiongwujie@gmail.com

Xinxin Fan
IoTeX
USA
fan.hecc@gmail.com

Lin Chen
Texas Tech University
USA
chenlin198662@gmail.com

Tianwei Zhang
Nanyang Technological University
Singapore
tianwei.zhang@ntu.sg.edu

Taeweon Suh
Korea University
South Korea
suhtw@korea.ac.kr

Weidong Shi
University of Houston
USA
wsh3@central.uh.edu

Veronika Kuchta
Florida Atlantic University
USA
vkuchta@fau.edu

Francesco Sica
Florida Atlantic University
USA
sicaf@fau.edu

Lei Xu
Kent State University
USA
xuleimath@gmail.com

Abstract

Cryptographic schemes like Fully Homomorphic Encryption (FHE) and Zero-Knowledge Proofs (ZKPs), while offering powerful privacy-preserving capabilities, are often hindered by their computational complexity. Polynomial multiplication, a core operation in these schemes, is a major performance bottleneck. While algorithmic advancements and specialized hardware like GPUs and FPGAs have shown promise in accelerating these computations, the recent surge in AI accelerators (TPUs/NPUs) presents a new opportunity. This paper explores the potential of leveraging TPUs/NPUs to accelerate polynomial multiplication, thereby enhancing the performance of FHE and ZKP schemes. We present techniques to adapt polynomial multiplication to these AI-centric architectures and provide a preliminary evaluation of their effectiveness. We also discuss current limitations and outline future directions for further performance improvements, paving the way for wider adoption of advanced cryptographic tools.

CCS Concepts

• **Security and privacy** → **Public key (asymmetric) techniques**; • **Computer systems organization** → *Special purpose systems*; • **Theory of computation** → *Mathematical optimization.*

ACM Reference Format:
Rabimba Karanjai, Sangwon Shin, Wujie Xiong, Xinxin Fan, Lin Chen, Tianwei Zhang, Taeweon Suh, Weidong Shi, Veronika Kuchta, Francesco Sica, and Lei Xu. 2024. TPU as Cryptographic Accelerator. In *International Workshop on Hardware and Architectural Support for Security and Privacy 2024 (HASP '24), November 02, 2024, Austin, TX, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696843.3696844

1 Introduction

In the last decades, a large number of cryptographic tools with fascinating features have been developed, such as fully homomorphic encryption (FHE) [4], zero-knowledge proof (ZKP) [8], and post-quantum cryptography schemes (PQC) [3]. FHE allows a third party to conduct arbitrary computation (addition and multiplication) on ciphertexts and ZKP helps a *prover* to convince a *verifier* on the correctness of a statement without disclosing the secret involved in the statement. These two types of cryptographic schemes can be utilized to build many interesting security applications, such as secure AI inferencing [7, 15] and privacy-preserving cryptocurrency exchange [32]. One major obstacle that hinders the wide adoption of these schemes is their high computation cost. PQC schemes are designed to resist attacks with quantum computers. They are not as expensive as FHE and ZKP, but all fall into the category of asymmetric cryptography (digital signature and public key encryption/key encapsulation mechanism) and are computation-intensive by nature.

Although these schemes are different from almost all perspectives like security features and constructions, they share one similarity in terms of computation, i.e., most of their operations involve polynomial calculations defined over specific algebra structures, which usually comprise a large portion of the computation cost of these schemes and the performance bottleneck. To fully utilize these cryptographic tools, it is important to improve the performance of the underlying basic polynomial operations, especially multiplication.

Many works have been done on improving the polynomial multiplication from the algorithmic perspective, such as Karatsuba multiplication [17] and number theoretic transform (NTT) based multiplication [21]. Implementation, especially hardware-based polynomial multiplication implementation, is another important research direction. The two major hardware platforms that are studied are GPU and Field Programmable Gate Array (FPGA). Both GPU and FPGA support parallel computing, which is utilized to accelerate polynomial multiplication [22, 31].

One emerging computation hardware that is largely ignored is the tensor processing unit (TPU), or neural processing unit (NPU). These two names usually refer to the same type of hardware, and we use the term TPU in the rest of the paper. The concept of TPU was originally proposed by Google to meet the explosive computation power demand of artificial intelligence (AI) applications and has been proven to be a big success [25]. Many TPU designs have been proposed since then, especially for the edge computing scenario. Compared with GPU and FPGA, TPU has its unique advantages. On the one hand, it is closer to Application-Specific Integrated Circuits (ASIC) and can significantly accelerate specific operations with low energy costs. On the other hand, TPU still provides a certain level of flexibility and allows end users to program it.

While TPUs are widely used in AI [10], their application in cryptography is largely unexplored. To the best of our knowledge, this is the first work investigating TPUs for polynomial multiplication, a core operation in FHE, ZKP, and PQC. To leverage TPUs' MACs, we convert polynomial multiplications to matrix multiplications. However, TPU's AI-centric design struggles with these matrices. We address this using a carefully chosen RNS [6] and matrix partitioning. Our prototype on Google's cloud TPU demonstrates TPU's promise for cryptographic acceleration, with significant room for improvement given the limitations of current TPU software tools.

From an algorithmic perspective, the polynomial multiplication method considered in this work uses coefficient representation and has higher computation complexity (about $O(n^2)$) than the point-value representation with FFT tricks (NTT, $O(n \log n)$). But it is still worth investigating polynomial multiplication with coefficient representation for at least two reasons: **(i)**Coefficient representation is a more convenient way to store polynomials and the multiplication is efficient when the degree is not very high; and **(ii)**NTT is only applicable for polynomials with certain features (e.g., the existence of nth root of unity in the finite field where the polynomial coefficients are defined and the degree is n).

In summary, the contributions of the paper include: **(i)** We design key technologies to enable the utilization of TPU for polynomial multiplications in a variety of cryptographic schemes; **(ii)** We develop a prototype using existing TPU hardware and software to demonstrate the feasibility of the design; and **(iii)** Last but not least, we discuss the potential ways to improve the TPU hardware and corresponding algorithms to further accelerate polynomial multiplication.

2 Detailed Design of TPU-based Polynomial Multiplication

Polynomials are used intensively in various public key cryptographic schemes mainly for two reasons: **(i)** Polynomial is related to popular hard problems used in cryptography. Many PQC and FHE schemes are built atop ring LWE problems [30], and using a polynomial ring is a major way to instantiate a ring LWE instance. **(ii)** Polynomial is an effective way for computation representation. An effective computation representation is critical for ZKP, and many ZKP schemes use polynomials in one way or another to represent the computation (or the statement) the prover needs to prove. Polynomial operations, especially polynomial multiplications, usually comprise a large portion of the computation cost of these cryptographic schemes and are the performance bottleneck.

Like a typical computation device, the TPU has four types of components: I/O, control, storage, and computation. The major part is a Matrix Multiply Unit, which is implemented as a systolic array. This unit can perform 256×256 multiply-and-accumulate (MAC) operations on signed or unsigned integers/floating point numbers, which are essential for matrix multiplications. In addition to the Matrix Multiplicity Unit, there is another useful computation unit, the vector unit, which is used for general computations such as the activation function.

The dedicated and optimized circuit is usually much faster and more power-efficient than CPU and GPU on MAC operations. According to [13], a Haswell server equipped with the original version of Google TPU can achieve 92 TOPS/s with a TDP of 861W, while the server itself can only achieve 2.6 TOPS/s with a TDP of 504W. After the first generation, Google has released three revised versions of TPU and the latest version is V4 [12].

2.1 Polynomials Used in Cryptography and Their Operations

Polynomials used in cryptography are usually defined over a ring structure $\mathbb{Z}_q[x]/(p(x))$. From the computation perspective, there are mainly three factors to consider: **(i)** *The degree of $p(x)$*. The degree of operand polynomials and operation result polynomials are less than the degree of $p(x)$. This is an important factor that dominates the complexity of the computation (addition and multiplication) in most cases. **(ii)***The value of q*. The value of q determines the range of polynomial coefficients. The size (i.e., the number of bits) and form (e.g., Hamming weight of the binary representation) of q determine the complexity of each operation on polynomial coefficients (e.g., addition and multiplication). **(iii)***The form of $p(x)$*. All computation results need to be reduced modulo $p(x)$, and the complexity of the modulo operation is highly related to the form of $p(x)$. Generally, the modulo operation is more complex when $p(x)$ has more non-zero coefficients.

In this work, we mainly consider the ring $\mathbb{Z}_q[x]/(x^n+1)$, which is widely used in the construction of different cryptographic schemes. Here n is a power of 2, making $x^n + 1$ a cyclotomic polynomial, and q is an integer and does not need to be a prime number, i.e. $\mathbb{Z}_q$ does not need to be a prime field. Another popular ring structure used in cryptography is $\mathbb{Z}_q[x]/(x^n - 1)$, but the technologies described in the rest of the paper can be easily adapted to handle the case of $\mathbb{Z}_q[x]/(x^n - 1)$.

A polynomial $f(x) \in \mathbb{Z}_q[x]/(x^n + 1)$ is in the form

$$f(x) = a_0x^0 + a_1x^1 + \cdots + a_{n-1}x^{n-1}, \qquad (1)$$

where $a_i \in \mathbb{Z}_q, i = 0, \dots, n-1$. The two operations defined on the ring are *polynomial addition* and *polynomial multiplication.* Polynomial addition is straightforward, i.e., two corresponding coefficients of two polynomials are added together and then reduced modulo q. Multiplication is more complex and consumes most of the computation time. We review two common ways to do multiplication, which are closely related to the representation of polynomials: **(i)** *Multiplication with coefficient representation.* Coefficient representation is a natural way to store a polynomial, as demonstrated in Eq. (1). When two polynomials defined in $\mathbb{Z}_q[x]/(x^n+1)$ are stored using coefficient representation, a straightforward approach is first multiplying them as if they are defined in $\mathbb{Z}[x]$, and then reducing the polynomial by taking modulo x^n+1 and q to obtain the final result. One disadvantage of this approach is that the intermediate result has a degree of at most $2n-2$ with coefficients as large as $(q-1)^2$. We may integrate these modulo operations into the multiplication to save some time. The complexity of a straightforward multiplication is $O(n^2)$, and using some techniques like Karatsuba algorithm can bring down the complexity to the level of $O(n^{\log_2 3})$ [17]. **(ii)** *Multiplication with value representation.* Another common way to store a polynomial is using value representation. For the polynomial $f(x)$ given in Eq. (1) represented with coefficients, one can evaluate it at n different values for x, and store all these n pairs $[x_1, y_1 = f(x_1)], [x_2, y_2 = f(x_2)], \dots, [x_n, y_n = f(x_n)]$. If we always use $x_i = i \in \mathbb{Z}, i = 1, 2, \dots, n$ for evaluating a polynomial, x_i can be ignored and only y_is need to be stored (we assume $n < q$, which usually holds). These pairs (or y_is) fully determine the polynomial and we can convert them back to coefficient representation using Lagrange interpolation. The multiplication of polynomials stored using value representation is easy, i.e., we only need to perform an element-wise multiplication. As most cryptography schemes use polynomials with coefficient representation, one needs to convert them to value representations first. A straightforward way of conversion is evaluating a polynomial at each given value, and the complexity of each evaluation is $O(n)$. Since the polynomial is evaluated at n values, the overall complexity is $O(n^2)$, which is more expensive than the multiplication operation in value representation. Because the multiplication result may need to be involved in other cryptographic operations, we usually need to convert it back to coefficient representation. This inverse operation can be done using straightforward Lagrange interpolation, and its complexity is also $O(n^2)$. When the two parameters q and n of $\mathbb{Z}_q[x]/(x^n+1)$ meet certain requirements, the conversion and reverse conversion can leverage number theoretic transform (NTT), and the complexity is reduced to the level of $O(n \log n)$.

Because of the lower computation complexity, multiplication using value representation receives a lot of attention. However, this approach has several limitations: **(i)** *Size of the problem.* The lower computation complexity does not guarantee faster execution in practice. NTT is faster only for relatively large polynomial degrees. When the cryptography schemes use polynomials with low degrees, the difference between these two multiplication methods is small, and value representation can be even slower. **(ii)***Restrictions on parameters of* $\mathbb{Z}_q[x]/(x^n+1)$. The low computation complexity of multiplication using value representation relies on the applicability of NTT and requires the polynomials to be NTT-friendly (e.g., n is a power of 2 and $q \equiv 1 \mod n$). While there are some works on losing these requirements [5], it is hard to get rid of them completely. These requirements restrict the parameter selection for the cryptography scheme, which is not desirable. **(iii)***Compatibility with the hardware processor.* Value representation-based multiplication can be implemented as software but it does not guarantee that the software can fully utilize the underlying hardware capability.

Considering all these factors, it is worth the effort to consider polynomial multiplication using coefficient representation. In this work, we focus on the polynomial multiplications defined in $\mathbb{Z}_q[x]/(x^n+1)$ using coefficient representation.

2.2 Converting Polynomial Multiplication to Matrix Operations

TPU is a special hardware that is designed to accelerate the inference operation in a deep neural network, especially the matrix operation. Therefore, we need to convert polynomial multiplications described in 2.1 to matrix multiplications to utilize the capability of the TPU hardware.

It is relatively easy to incorporate the reduction of modulo x^n+1 into the multiplication process itself by modifying one of the operand polynomials. Example 1 gives a toy example of incorporating the modulo operation into polynomial $b(x)$.

Example 1. Considering two degree-2 polynomials defined over $R_q = \mathbb{Z}_q[x]/(x^3+1)$, $a(x) = a_0x^0 + a_1x^1 + a_2x^2$ and $b(x) = b_0x^0 + b_1x^1 + b_2x^2$. The multiplication process can be converted to a vector-matrix multiplication operation as follows:

$$(a_0, a_1, a_2) \times \begin{bmatrix} b_0 & b_1 & b_2 \\ -b_2 & b_0 & b_1 \\ -b_1 & -b_2 & b_0 \end{bmatrix} = \begin{bmatrix} a_0b_0 - a_1b_2 - a_2b_1 \\ a_0b_1 + a_1b_0 - a_2b_2 \\ a_0b_2 + a_1b_1 + a_2b_0 \end{bmatrix}. \quad (2)$$

The vector on the right side of Eq. (2) presents the multiplication result, i.e., $(a_0b_0 - a_1b_2 - a_2b_1)x^0 + (a_0b_1 + a_1b_0 - a_2b_2)x^1 + (a_0b_2 + a_1b_1 + a_2b_0)x^2$. Note that all coefficients here need to take modulo q. The matrix on the left side of Eq. (2) is determined by the modular polynomial x^n+1 and the polynomial $b(x)$. □

When the modulo polynomial is set to x^n+1, the modulo operation is simpler than a general modulus polynomial. Without loss of generality, we assume the original multiplication result is $\sum_{i=0}^{2n-2} c_i x^i$. For a term $c_d x^d$ in the multiplication result with a degree $d \geq n$, we subtract the coefficient c_d from the $(d-n)$th term, i.e., $(c_{d-n} - c_d)x^{d-n}$, to obtain the reduced result. Following this idea, we can convert the multiplication into matrix multiplication format with moduli polynomial x^n+1 as follows:

$$(a_0, a_1, \dots, a_{n-1}) \times \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \\ -b_{n-1} & b_0 & \cdots & b_{n-2} \\ \cdots & \cdots & \cdots & \cdots \\ -b_1 & -b_2 & \cdots & b_0 \end{bmatrix}. \quad (3)$$

For the matrix in Eq. (3), each row (except the first one) is a right shift of all elements below the main diagonal multiplied by -1.

The computation given in Eq. (3) is actually a vector-matrix multiplication, and it can be easily converted to matrix-matrix multiplication when one fixed polynomial (the matrix) is multiplied

Table 1: Common polynomial parameters for cryptographic schemes.

Scheme	Polynomial degree	Polynomial coefficients size
FHE (FV, BFV, CKKS)	2^{10} to 2^{14}	32 to 54 bits
PQC	2^8 to 2^{10}	$\leq$ 60 bits
ZKP (zkSNARK, zkSTARK)	2^{20} to 2^{21}	384 to 768 bits

with many different polynomials, where each of them is represented as a vector and all the vectors (a row) form a matrix.

In other words, multiple polynomial multiplications can be converted to a single matrix multiplication, as long as one of the operand polynomials is the same.

2.3 Challenges of Using TPU for Polynomial Multiplication

The way of converting polynomial multiplication to matrix operation described above is not enough for utilizing TPU for cryptographic operations. The first challenge is the degree of the polynomials. For polynomials with degree n, the corresponding matrix dimension is $n \times n$. The degree of the polynomials involved in a typical cryptographic scheme can be large, and the corresponding matrix cannot fit the TPU hardware for computation. The second challenge is the size of the polynomial coefficients. Most TPUs are designed for machine learning tasks, and a significant amount of work has been done to reduce the size of values involved in a machine learning model. For instance, the Google TPU supports `bf16` natively, which is a 16-bit floating point number system and has lower precision compared with the standard 32-bit floating point number, but provides similar model performance [28]. However, polynomials used in cryptography usually have relatively large coefficients, from tens of bits to a few hundred bits. Furthermore, these coefficients are generally defined on a finite and discrete algebra structure, which is not natively consistent with the data type supported by the TPU. Table 1 summarizes some polynomial parameters used by different cryptographic schemes. We describe the way to overcome these two challenges in the following sections.

2.4 Handling Large Coefficients

Classical digital computers store and process values presented using a limited number of bits. For large values, there are two strategies to process them: **(i)***Using an approximated value to replace the original one.* With this approach, limited storage is used to store a value that is close to the real value. Floating point number schemes like `IEEE 754` [14] and `bfloat16` [28] fall into this category of solution. This strategy is effective for a wide range of applications where the problems do not require 100% computation accuracy. **(ii)***Using multiple basic units to represent the original one.* The other one is breaking a large value into several parts so each part can be stored and processed by the hardware. Accordingly, an operation on the original large values is also decomposed into a sequence of operations on the small units, which can be handled by the hardware directly. One common example is big integer operations, where an integer with more than 32 or 64 bits is stored in multiple basic data structures, and dedicated algorithms are implemented to support the common arithmetic operations [2]. This strategy is widely used in cryptographic libraries to support schemes such as RSA.

Since cryptographic operations usually deal with accurate and discrete values, the first approach is not a good option. Applying the second strategy directly to the method described 2.2 is not ideal either, as each coefficient multiplication/addition needs to be decomposed into several operations, which cannot be carried out by the TPU directly. This will lead to more complex design and lower performance.

To overcome this challenge, we propose to utilize the residue number system (RNS) [27] for coefficients representation and computation. A residue number system is defined by a set of k co-prime integers called the moduli $\{m_1, m_2, \dots, m_k\}$. An integer x is represented in the RNS by its remainder $[x_1, x_2, \dots, x_k]$, where $x_i \equiv x \mod m_i$ for every i.

For addition/subtraction and multiplication operations of numbers represented in an RNS, it suffices to perform the same operation on each pair of residues. For two integers represented in RNS, i.e., $x = [x_1, x_2, \dots, x_k]$ and $y = [y_1, y_2, \dots, y_k]$, their product z is in the form $z = [z_1, z_2, \dots, z_k]$, where $z_i = x_i \times y_i \mod m_i$ for every i.

To recover z from $[z_1, z_2, \dots, z_k]$, we can use the Chinese Reminder Theorem since all moduli are selected to be co-prime. Note that we can keep the RNS representation until there is a need to convert the result back to $\mathbb{Z}_q[x]/(x^n + 1)$.

There are two advantages of using RNS to handle large coefficients in our case: **(i)** The original computation is divided into multiple instances with small values and these new instances are independent, i.e., they can be executed in parallel. **(ii)** Each instance can be executed using the same framework described in 2.1, and does not require any modification.

2.5 Handling High Polynomial Degrees

The TPU matrix multiplication hardware has a limited size and can only handle matrices efficiently within such limitations. While it may handle much larger matrices with the help of corresponding tool chain software, the overall performance can decrease significantly. For a polynomial with a high degree (e.g., 2^{20}), it is hard to store the corresponding matrix completely in TPU memory. To overcome this challenge, we adopt the divide-and-conquer method, i.e., dividing the original matrix/vector into several smaller ones that can be processed by the TPU hardware directly, computing several intermediate results, and reconstructing the final result from these intermediate values.

Fortunately, we can naturally divide polynomial multiplication with a large degree into several smaller instances. Without loss of generality, we consider the case where we need to shrink the degree of polynomials by half, i.e., instead of processing degree n polynomials multiplication, we want to break it to operations on polynomials of degree $n/2$. The vector of dimension n (corresponding to one degree n polynomial operand) is directly divided into two vectors of dimension $n/2$. The matrix of dimension $n \times n$ (corresponding to the other degree n polynomial operand) is divided into four sub-matrices, and each of them has dimension $n/2 \times n/2$.

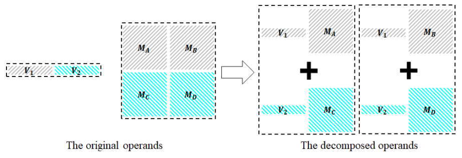


Figure 1: Demonstration of handling of polynomials with high degree. On the left side of the figure, we evenly break the vector into two sub-vectors and the matrix into four sub-matrices. On the right side, the original vector-matrix multiplication is decomposed into four vector-matrix multiplications with smaller dimensions.

Figure 1 depicts the way of breaking high-degree polynomials multiplication to low-degree polynomials calculations. To calculate and merge the intermediate results, we:

(1) Calculate the product of sub-vectors and sub-matrices:

$$r_{1A} \leftarrow V_1 \times M_A \qquad r_{1B} \leftarrow V_1 \times M_B$$
$$r_{2C} \leftarrow V_2 \times M_C \qquad r_{2D} \leftarrow V_2 \times M_D$$

All results are vectors of dimension $n/2$.

(2) Calculate the first half of the final result $r_1 = r_{1A} + r_{2C}$, which is a vector of dimension $n/2$.

(3) Calculate the second half of the final result $r_2 = r_{1B} + r_{2D}$, which is a vector of dimension $n/2$.

(4) Concatenate r_1 and r_2 to form the final result, which is a vector of dimension n.

The dividing process can be repeated recursively until the vector/matrix can fit the TPU hardware. While this approach can reduce the dimension of vector/matrix, it also increases the number of operations. In the above case, one single vector-matrix multiplication becomes four smaller vector-matrix multiplications with some extra operations.

While this work primarily focuses on polynomial multiplication in the ring $\mathbb{Z}_q[x]/(x^n+1)$, the core ideas and techniques presented can be extended to other ring structures and polynomial forms commonly encountered in cryptographic applications. The conversion of polynomial multiplication to matrix multiplication, as described in Section 2.3, relies on the structure of the polynomial modulus. For different moduli, such as $x^n - 1$ or trinomials, the corresponding matrix structure will change, potentially requiring adjustments to the mapping process. However, the fundamental principle of leveraging the TPU's matrix multiplication capabilities remains applicable. Furthermore, the use of the Residue Number System (RNS) for handling large coefficients is independent of the specific ring or polynomial form. Future work will explore these generalizations in detail, including adapting the matrix conversion algorithm for different moduli and analyzing the performance impact on various cryptographic protocols with diverse polynomial structures.

When the chosen number of moduli (m) results in matrices that exceed the TPU's size limitations, we adopt a hierarchical divide-and-conquer strategy. The large matrices are further subdivided into smaller blocks that fit within the TPU's memory constraints. This involves performing multiple smaller matrix multiplications and then combining the intermediate results to obtain the final product. This approach increases the number of matrix multiplications but allows us to handle larger RNS bases and potentially improve performance by increasing parallelism. The overhead of this hierarchical division is minimized by careful scheduling and data management within the TPU environment.

2.6 Key Parameters Selection

For polynomial multiplication in cryptography, two important parameters are associated with large coefficients and high-degree handling. One parameter is the size of the RNS base (and corresponding moduli), and the other parameter is the matrix dimension, which determines the size of the basic matrix calculation.

RNS base selection. Given the ring $\mathbb{Z}_q[x]/(x^n + 1)$, there are several requirements for RNS base selection. **(i)** Each modulus of the RND base should fit the TPU hardware so related calculations can be done easily. **(ii)** The product of all moduli of the RNS base should be larger than q^2 if we only need to do one multiplication operation with the given RNS system and convert it back to its original form. As we need to compute matrix multiplication, several intermediate products are added together in the computation process, so we require the product of the RNS base to be larger than ℓq^2, where ℓ is the number of addition operations. At the same time, we prefer an RNS base with fewer members to reduce computation costs, so a larger modulus should be chosen first.

The selection of a suitable RNS base involves carefully considering several factors. First, the word length of the moduli is chosen to be as close as possible to the TPU's maximum integer word length, which is **8 bits** for the Google TPU v2 and v3 used in our experiments. This maximizes the utilization of the TPU's arithmetic capabilities and minimizes the overhead of splitting larger coefficients. Second, the number of moduli (m) is chosen to balance the benefits of parallelism with the overhead of conversion and reconstruction. We aim for an m that provides sufficient parallelism without causing excessive memory consumption or communication overhead. Third, the moduli are selected to be co-prime to ensure correct reconstruction using the Chinese Remainder Theorem (CRT). Their product must be larger than the maximum possible value of the coefficients after multiplication, which depends on the value of q and the degree of the polynomials. In our experiments, for a given m, we choose the largest possible co-prime moduli that fit within the 8-bit word length. Finally, we prioritize moduli that allow for efficient modular arithmetic on the TPU. For instance, moduli close to powers of two can simplify modulo operations and improve performance.

The value of q in the polynomial ring $\mathbb{Z}_q[x]/(x^n + 1)$ and the TPU's maximum word length are crucial factors in determining the efficiency of the RNS representation and the overall performance. In our experiments, we consider values of q that are larger than the 8-bit word length of the TPU. This requires representing coefficients using multiple moduli in the RNS, effectively performing multi-precision arithmetic within each modular computation. This introduces a potential bottleneck, as the TPU is optimized for single-word integer operations. However, by carefully selecting the RNS base and utilizing efficient modular arithmetic techniques, we aim to minimize the overhead of multi-precision computations. Future work will explore optimizations such as incorporating Montgomery or Barrett reduction algorithms to further improve the performance of these operations.

Matrix dimension selection. The matrix dimension selection also relies on the structure of $\mathbb{Z}_q[x]/(x^n+1)$ and the TPU hardware parameters. We prefer to use a larger dimension when possible as it can simplify the implementation and reduce the number of matrix multiplications. However, a large dimension may cause two issues: **(i)** The TPU does not have enough memory to store the matrices, which can crash the application. We have observed crashes for multiple Google Cloud TPUs with large matrices. **(ii)** The dimension does not fit the TPU directly, and the system needs to take extra steps to break it down and the efficiency is not guaranteed.

While a simple approach sets the dimension to the matrix multiplier's size, cloud TPUs often have multiple multipliers, making fine-tuning challenging. Experimentation is key to finding the optimal dimension, requiring more effort than RNS base selection.

2.7 Putting All Pieces Together

Having described the conversion of polynomial multiplication to matrix multiplication and addressed the challenges of large coefficients and degrees, we now outline the process of utilizing TPUs for polynomial multiplication in cryptographic schemes. The algorithm comprises three main steps: (i) determining the parameters to decompose the original polynomials for TPU compatibility; (ii) computing intermediate multiplication results on the TPU; and (iii) reconstructing the final polynomial multiplication results from these intermediate values. This process is summarized in Algorithm 1.

Algorithm 1 Polynomial multiplication using TPU.

```
Input: Two polynomials $a(x), b(x) \in \mathbb{Z}_q[x]/(x^n+1)$, TPU parameters $para$
Output: $c(x) = a(x)b(x) \in \mathbb{Z}_q[x]/(x^n+1)$
    ▷ Creating the configuration information based on the problem and the hardware
1: $config \leftarrow$ GenConfig($(\mathbb{Z}_q[x]/(x^n+1), para)$)
    ▷ Extract the list of moduli from $config$, which is used to build the RNS base
2: $modList \leftarrow$ ExtractModuli($config$)
    ▷ Extract matrix dimension information from $config$
3: $dimList \leftarrow$ ExtractDimensions($config$)
    ▷ Use selected parameters to convert polynomials to matrix form for multiplication on TPU
4: $A \leftarrow$ ConvertPoly($a(x), modList, dimList$)
5: $B \leftarrow$ ConvertPoly($b(x), modList, dimList$)
    ▷ Calculate matrix multiplication using TPU
6: $C \leftarrow$ TPUMul($A, B$)
    ▷ If needed, convert back the multiplication result to coefficient representation
7: $c(x) \leftarrow$ ConvertMatrix($C, modList, dimList$)
8: return $c(x)$
```

3 Experiments and Evaluation

Due to the rising popularity of AI applications [16], there is a growing interest in the design and development of TPU hardware [26]. However, the number of TPU products accessible to general developers remains limited. One of the most widely used options is Google's TPU, which is what we primarily evaluate our design using.

Table 2: Benchmarking environments on Google Cloud.

Env	Description (Physical Cores)
Env 1 (Figure 2a)	Ubuntu 22.04, TPU v2-8: 8 cores, 64 GB memory.
Env 2 (Figure 2b)	Ubuntu 22.04, TPU v3-8: 8 cores, 128 GB memory.

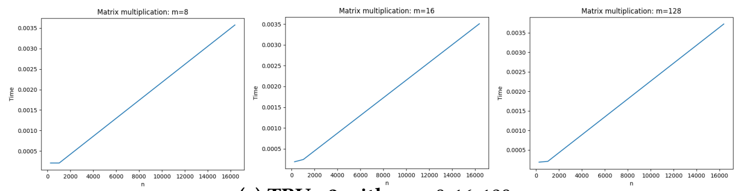
Overview of Google Cloud-based TPU. Google offers two primary ways to access TPUs: via a TPU-equipped virtual machine (VM) or through Colab, an online platform with TPU connectivity. Both support frameworks like TensorFlow, PyTorch, and JAX, which are Python-based and designed for AI applications. We opted for a TPU VM with JAX, prioritizing its flexibility in configuring hardware and support for non-AI computations and performance optimization. Colab, while convenient, offers limited hardware configuration and currently lacks support for the JAX version used in our experiments.

Prototype and evaluation. The most computation-intensive part of polynomial multiplication in $\mathbb{Z}_q[x]/(x^n+1)$ is the matrix multiplication operation. As we discussed in Section 2, the size of coefficients and the degree of the polynomial will affect the dimensions of the matrix we have to deal with. We evaluate the performance using three polynomial degrees, i.e., $n = 2^8, 2^{10}, 2^{14}$. For the RNS base size, we also consider three values, i.e., $m = 8, 16, 128$. Table 2 describes the two benchmarking environments.

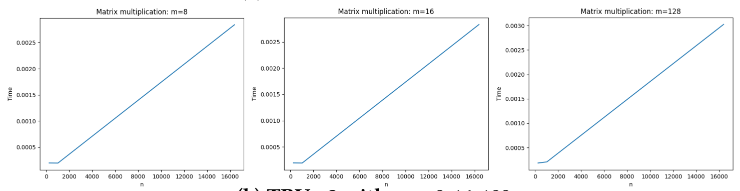
Figure 2a and Figure 2b summarize the benchmarking results using two execution environments Env 1 and Env 2 respectively. For each figure, we fix the value of m and the x-axis is different values of n. The y-axis is the execution time and the unit is the second. We observe several things from the experiment results: **(i)** For small matrix dimensions, the computation time is a constant value. After the threshold, the computation time increases linearly with the value of n. Note that the overall computation complexity is $O(m \times n \times n)$ (if the computation is done using the straightforward algorithm), which is not consistent with the experiment results. One potential reason is that the TPU architecture carries out the computation in a highly parallel manner and reduces the latency. **(ii)**The slopes of curves for different m values are similar, which means that m does not affect the cost much. The potential reason is that the TPU hardware supports 128×128 matrix multiplication, and the largest m we choose is within this range. **(iii)** TPU v3 is faster than TPU v2, but the difference is not significant, especially for cases with small parameters. This may be because both versions have the same number of matrix multiplication hardware (Table 2).

Limitations of the current benchmarking.

While our experiments demonstrate the potential of TPUs as cryptography accelerators, our benchmarking is limited by the inability to fully control execution granularity, potentially underutilizing TPU capabilities. For example, despite the larger memory capacity of TPU v3, the current programming environment lacks direct control over its usage. Ideally, pipelining multiple matrix multiplications could reduce latency, but Google's TPU framework currently lacks support for such fine-grained optimization.



(a) TPU v2 with $m = 8, 16, 128$



(b) TPU v3 with $m = 8, 16, 128$

Figure 2: Summary of experiment results using Google TPU with different configurations.

4 Further Improving Polynomial Multiplication with TPU

Having addressed the feasibility of accelerating polynomial multiplication on TPUs, we now turn our attention to optimization techniques that further enhance system performance.

Karatsuba multiplication. The idea of Karatsuba multiplication [17] is trading one expensive multiplication with multiple cheap addition/subtraction operations. For two polynomials $f(x)$ and $g(x)$ with degree n, we can rewrite them as $f(x) = f_1(x)x^m + f_0(x)$ and $g(x) = g_1(x)x^m + g_0(x)$, where f_1, g_1 are polynomials of degree $n - m$, and f_0, g_0 are polynomials of degree $m - 1$. The product is then

$$\begin{aligned} f(x)g(x) &= (f_1(x)x^m + f_0(x))(g_1(x)x^m + g_0(x)) \\ &= h_2(x)x^{2m} + h_1(x)x^m + h_0(x), \end{aligned}$$

where $h_2(x) = f_1(x)g_1(x), h_1(x) = (f_1(x)+f_0(x))(g_1(x)+g_0(x)) - h_2(x) - h_0(x), h_0(x) = f_0(x)g_0(x)$.

This technique can also be applied to accelerate TPU-based polynomial multiplication. Specifically, the original polynomials are decomposed into multiple lower-degree polynomials, and the TPU is then employed to perform the multiplications. This method can be used in place of the high-degree polynomial handling approach described in 2.5. However, a key challenge lies in the TPU's inefficiency for polynomial addition and subtraction. The use of Karatsuba multiplication may require additional hardware, introducing communication overhead and complicating optimization.

Fine-tuning the design. In this work, we primarily address the feasibility of using TPUs for polynomial multiplication. In practice, cryptographic schemes often require multiple polynomial multiplications in a specific sequence. Several strategies can be considered to fine-tune the computation: **(i)** *Fixing one operand polynomial as much as possible.* One operand polynomial needs to be converted to matrix form to incorporate the modulo operation into the multiplication. Creating the matrix and loading it into TPU memory is costly compared to matrix multiplication itself. Therefore, reusing a pre-generated and loaded matrix minimizes the preparation overhead. **(ii)** *Optimizing computation and I/O.* Polynomial multiplication in cryptography is both computation- and I/O-intensive, and the environment is complex. As TPUs function as passive accelerators, data must be transferred from the host machine to the TPU card. Since a machine typically has multiple TPUs, selecting the appropriate TPU at a given time is crucial. Moreover, intermediate results generated by the TPU might need to be sent back to the host for further processing, so ensuring the TPU remains occupied with other tasks during this transfer is essential.

5 Related Works

Accelerating multiplication of polynomials defined in specific rings has been studied from different perspectives.

Algorithmic approaches. Multiplication acceleration methods are studied for both coefficient and value representations, e.g., Karatsuba algorithm [17] and various NTT approaches [5]. Some works also consider more efficient modulo operation, which is heavily involved in polynomial operations [29]. These works are orthogonal to our work and have the potential to be utilized to further improve the polynomial multiplication performance with the TPU.

Hardware-oriented optimization approach. This line of research investigates the utilization of existing hardware features to accelerate polynomial multiplication on specific rings. Several types of hardware are studied for this purpose, including CPU [1], GPU [24], and FPGA [23]. Compared with other types of hardware, TPU provides a new balance between flexibility, power consumption, and parallelism. Table 3 compares TPU with other computation platforms.

Table 3: Comparison of major computation platforms.

Hardware	Flexibility	Power Consumption	Parallelism
CPU	Very High	High	Low
GPU	High	Very High	High
FPGA	Moderate	Low	High
TPU	Low	Low	High

TPU related works. Most existing works on TPU focus on its architecture [11, 13] and applications in AI [18, 19], with a few exceptions considering TPU for non-AI use cases. For instance, Hsu and Tseng developed a tool to use edge TPU for general-purpose computation [9], and Lewis et.al discussed the use of TPU for large-scale linear algebra computation [20]. None of these works consider the special requirements of cryptographic schemes.

6 Conclusion and Future Work

This work explores the acceleration of polynomial multiplication, a crucial operation in various cryptographic schemes, using Tensor Processing Units (TPUs). To overcome challenges in managing large coefficients and high-degree polynomials on TPUs, we employ the Residue Number System (RNS) and a divide-and-conquer approach, respectively. Future work will focus on applying TPUs for NTT-based polynomial multiplication, optimizing hardware configurations for specific polynomial rings, and designing end-to-end cryptographic schemes with TPUs, incorporating optimization strategies.

Acknowledgments

This research was partially supported by NATO SPS G7200, NCAE H98230-24-1-0097, and Google TRC and GDE. We thank these organizations for their funding and support.

References

[1] Carlos Aguilar-Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrede Lepoint. 2016. NFLlib: NTT-based fast lattice library. In *Topics in Cryptology-CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29-March 4, 2016, Proceedings.* Springer, 341–356.

[2] Rajorshi Biswas, Shibdas Bandyopadhyay, and Anirban Banerjee. 2003. A fast implementation of the RSA algorithm using the GNU MP library. In *IIIT-Calcutta, National workshop on cryptography*.

[3] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 353–367.

[4] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical Gap-SVP. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings.* Springer, 868–886.

[5] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 159–188.

[6] Harvey L Garner. 1959. The residue number system. In *Papers presented at the the March 3-5, 1959, western joint computer conference.* 146–153.

[7] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.

[8] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *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 35.* Springer, 305–326.

[9] Kuan-Chieh Hsu and Hung-Wei Tseng. 2021. Accelerating applications using edge tensor processing units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–14.

[10] Fantine Huot, Yi-Fan Chen, Robert Clapp, Carlos Boneti, and John Anderson. 2019. High-resolution imaging on TPUs. *arXiv preprint arXiv:1912.08063* (2019).

[11] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture.* 1–14.

[12] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. 2021. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 1–14.

[13] Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. A domain-specific architecture for deep neural networks. *Commun. ACM* 61, 9 (2018), 50–59.

[14] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[15] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. Scaling up trustless dnn inference with zero-knowledge proofs. *arXiv preprint arXiv:2210.08674* (2022).

[16] Rabimba Karanjai, Lei Xu, and Weidong Shi. 2024. SolMover: Smart Contract Code Translation Based on Concepts. In *Proceedings of the 1st ACM International Conference on AI-Powered Software.* 112–121.

[17] Anatolii Alekseevich Karatsuba and Yu P Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.

[18] Haklin Kimm, Incheon Paik, and Hanke Kimm. 2021. Performance comparision of TPU, GPU, CPU on Google colaboratory over distributed deep learning. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC).* IEEE, 312–319.

[19] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. 2019. Scale mlperf-0.6 models on google tpu-v3 pods. *arXiv preprint arXiv:1909.09756* (2019).

[20] Adam GM Lewis, Jackson Beall, Martin Ganahl, Markus Hauru, Shrestha Basu Mallick, and Guifre Vidal. 2022. Large-scale distributed linear algebra with tensor processing units. *Proceedings of the National Academy of Sciences* 119, 33 (2022), e2122762119.

[21] Zhichuang Liang and Yunlei Zhao. 2022. Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey. *arXiv preprint arXiv:2211.13546* (2022).

[22] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2019. Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD).* IEEE, 253–260.

[23] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2020. FPGA implementation of a run-time configurable NTT-based polynomial multiplication hardware. *Microprocessors and Microsystems* 78 (2020), 103219.

[24] Ning Ni and Yongxin Zhu. 2023. Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU. *J. Parallel and Distrib. Comput.* 173 (2023), 20–31.

[25] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The design process for Google's training chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63.

[26] Eric B Olsen. 2017. Proposal for a high precision tensor processing unit. *arXiv preprint arXiv:1706.03251* (2017).

[27] Amos R Omondi and A Benjamin Premkumar. 2007. *Residue number systems: theory and implementation.* Vol. 2. World Scientific.

[28] John Osorio, Adrià Armejach, Eric Petit, Greg Henry, and Marc Casas. 2022. A BF16 FMA is all you need for DNN training. *IEEE Transactions on Emerging Topics in Computing* 10, 3 (2022), 1302–1314.

[29] Karl C. Posch and Reinhard Posch. 1995. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems* 6, 5 (1995), 449–454.

[30] Miruna Rosca, Damien Stehlé, and Alexandre Wallet. 2018. On the ring-LWE and polynomial-LWE problems. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 146–173.

[31] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2022. Accelerating polynomial multiplication for homomorphic encryption on gpus. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED).* IEEE, 61–72.

[32] Lei Xu, Lin Chen, Zhimin Gao, Keshav Kasichainula, Miguel Fernandez, Bogdan Carbunar, and Weidong Shi. 2020. PrivateEx: privacy preserving exchange of crypto-assets on blockchain. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing.* 316–323.