SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition

Cheng Wang[1,3†] and Mingyu Gao[2,3,4*]

[1]Xi'an Jiaotong University
[2]Tsinghua University
[3]Institute for Interdisciplinary Information Core Technology, Xi'an
[4]Shanghai Qi Zhi Institute

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323744

***Abstract*—With an increasing focus on data security in today's computer systems, homomorphic encryption and zero-knowledge proofs are becoming widely used tools in privacy-preserving computing. Number theoretic transform (NTT) is a key primitive that dominates the performance of these algorithms, and thus becomes an attractive target for domain-specific acceleration. Prior NTT accelerators mostly support only fixed and small NTT sizes, which are insufficient for the diverse parameter requirements of different cryptographic algorithms and applications.**

In this paper, we propose an FPGA-based, scalable NTT accelerator that uses multi-dimensional decomposition to efficiently support various NTT sizes. The hardware uses a limited and fixed amount of compute and storage resources on-chip. An arbitrary-sized NTT task is decomposed into fixed-sized small NTT kernels that match the on-chip hardware and thus execute with high efficiency. We further incorporate techniques to optimize both off-chip and on-chip data transfers under such complicated decomposed execution. Overall, our accelerator balances between on-chip compute throughput and off-chip memory bandwidth. It can flexibly scale to very large NTT tasks, and outperforms prior FPGA-based NTT accelerators by over $2\times$ at these large sizes.

***Index Terms*—NTT, accelerator, FPGA**

I. Introduction

Data privacy has recently become a critical concern in computer systems due to the ubiquity of cloud computing, blockchains, and Internet of Things (IoT). To guarantee confidentiality and integrity not only when data are stored and transmitted but also during computation, modern cryptography offers various privacy-preserving computing algorithms, such as homomorphic encryption (HE) and zero-knowledge proof (ZKP). HE [1] allows to encrypt the private data and conducts various computations on the ciphertexts, to directly obtain the encryption of the desired result without exposing the secret plaintexts, thus ensuring *confidentiality*. ZKP [2], on the other hand, allows a prover to generate a proof about some statement on her secret data x, such as $F(x) = 0$, while the proof is zero-knowledge, i.e., not leaking any information about the secret x when validated by other verifiers. ZKP is widely used in blockchains and computation outsourcing to ensure *integrity*.

State-of-the-art HE and ZKP algorithms typically use some modular polynomial rings as their underlying algebraic systems [2]–[7], and perform a large number of polynomial additions and multiplications. In particular, with a large polynomial degree N, the multiplication is an expensive operation, whose cost grows as $O(N^2)$. The most widely used algorithmic optimization for polynomial multiplications is based on the number theoretic transform (NTT), a special form of the discrete Fourier transformation on large finite fields. NTT could reduce the multiplication complexity to $O(N \log N)$. Nevertheless, it still remains as the dominant component in the overall performance of most HE and ZKP algorithms. Therefore, NTT has been a popular acceleration target using FPGA [8]–[15], ASIC [16]–[19], and GPU [20]–[22].

However, the algorithm parameters of NTT, particularly the polynomial degree N and the bitwidth W, differ significantly not only between HE and ZKP, but also across different protocols in each of the two domains. Most HE schemes [23]–[25] work on numbers with W of 50 to 60 bits after bit decomposition. Their polynomial degrees vary around 2^{16} to 2^{18}. In contrast, ZKP protocols [26], [27] require much larger polynomial degrees over 2^{20} and up to 2^{28}, and have to compute on the original wide integers of several hundreds of bits, e.g., $W = 256$. An ideal NTT accelerator should support these diverse ranges of algorithm parameters. But unfortunately, most previous FPGA-based and ASIC-based NTT accelerators were designed for a fixed algorithmic configuration, or only worked efficiently within a narrow range of parameters.

In this work, we propose SAM, an FPGA-based, *scalable* NTT accelerator that uses *multi-dimensional decomposition* to efficiently support various NTT sizes. Particularly, SAM uses the *same* hardware design with a *limited and fixed* amount of on-chip resources, to efficiently process NTT kernels of various sizes N, ranging from 2^{16} in HE up to 2^{28} in ZKP. It does this by decomposing the large NTT workload into a multi-dimensional hypercube of fixed and small NTT kernels, so the hardware only needs to use constant resources to implement efficient NTT units of fixed and small sizes. SAM then leverages FPGA's reconfigurability to instantiate arithmetic units for the specific bitwidth W, and seamlessly plugs them into the same high-level architecture without redesign efforts.

We emphasize that, although previous NTT accelerators have also used multi-dimensional decomposition [16]–[19], SAM is the first design that uses it to scalably support diverse

[†] Work done during the internship at Institute for Interdisciplinary Information Core Technology, Xi'an.
[*] Corresponding author.

NTT sizes. Specifically, previous designs decomposed a size-N NTT into *fixed numbers of dimensions* (e.g., 2-D or 3-D). When N varies, the decomposed small NTT sizes also change, either underutilizing the provisioned resources, or exceeding the hardware limit and not able to run. SAM instead fixes the post-decompose NTT sizes (and thus efficiently utilizing the hardware resources), and allows to decompose into *variable numbers of dimensions*. This is enabled by its novel computation flow and control logic design (Section III-B).

To better support the complicated execution of multi-dimensional NTT decomposition, SAM adopts novel microarchitectural optimizations. First, for off-chip data accesses, SAM generates the NTT twiddle factors on-the-fly instead of always accessing them from memory to save bandwidth. The scheme is specially tailored to decomposed NTT computations that are not supported before (Section III-C). Second, for on-chip data transfers, SAM proposes a novel data layout to rearrange data among multiple on-chip buffers to avoid buffer access conflicts while still preserving sufficiently sequential off-chip data fetches even though the computation is decomposed into small chunks (Section III-D). Finally, SAM also uses more area-efficient modular multipliers co-designed with the NTT pipeline structure, and a prefetching mechanism to improve off-chip bandwidth utilization (Section III-E).

To evaluate SAM, we instantiate two optimized configurations for 256-bit and 64-bit bitwidths, respectively, through careful design space exploration that achieves balanced computation throughput and data bandwidth (Section IV). Compared to the state-of-the-art FPGA-based NTT accelerators with similar off-chip memory bandwidth, SAM achieves 2.1× to 2.6× speedups for large NTT sizes of 2^{20} to 2^{24}. It also slightly outperforms several FPGA baselines at relatively small NTT sizes of 2^{16} and 2^{17}, even if these designs are specially optimized for these sizes and keep all data fitting on-chip.

II. Background

Number theoretic transform (NTT) is a number-theory-based form of discrete Fourier transform (DFT) that works on a finite field such as $\mathbb{Z}_p$ (integers modulo p), rather than on the complex plane. The modular operations in NTT avoid potential floating-point accuracy errors in traditional DFT, making NTT a powerful cryptographic primitive. In this work, we aim to design an FPGA-based accelerator for NTT.

A. Number Theoretic Transform (NTT)

Let $A(x) = \sum_{j=0}^{N-1} a_j x^j$ represent a polynomial with the degree of $N-1$. We can also view A as a vector of length N, and we use $a[j]$ to denote each of its elements, which is W-bit. We assume N is a power of 2; for non-power-of-2 lengths, we pad the vector with zeros. The NTT computation, $\hat{A} = \mathsf{NTT}(A)$, is mathematically defined as the following equation:

$$\hat{a}[i] = \sum_{j=0}^{N-1} a[j] \cdot \omega_N^{ij}, \qquad i = 0, \ldots, N-1$$

where ω_N is the Nth root of unity in the finite field. Any power of ω_N is called a *twiddle factor*. The multiplications and additions in the equation are also performed in the specific field, e.g., in $\mathbb{Z}_p$ as modular multiplications and modular additions. We call A and $\hat{A}$ respectively the *coefficient representation* and the *NTT representation* of the polynomial $A(x)$.

As a special form of DFT, NTT can also leverage fast Fourier transform (FFT) to reduce the computational complexity from $O(N^2)$ to $O(N \log N)$. An NTT with N input elements needs $\log_2 N$ stages where each stage has $N/2$ butterfly operations. Each butterfly operation takes two elements with a certain stride ($\frac{N}{2^i}$ for the ith stage), and generates two output elements using a modular addition, a modular subtraction, and a modular multiplication in $\mathbb{Z}_p$. For the ith stage, $\frac{N}{2^i}$ different twiddle factors are needed. With a fixed N and a fixed finite field, these twiddle factors are constant numbers.

To convert the NTT representation back to the coefficients, the inverse NTT (iNTT) process is similar to NTT, except that ω_N is replaced by ω_N^{-1} and the result elements are multiplied by N^{-1} at the end, i.e.,

$$a[j] = N^{-1} \cdot \sum_{i=0}^{N-1} \hat{a}[i] \cdot (\omega_N^{-1})^{ij}, \qquad j = 0, \ldots, N-1$$

B. Applications of NTT

NTT and iNTT are widely used in many modern cryptographic algorithms, such as homomorphic encryption (HE) [1], [3]–[5] and zero-knowledge proof (ZKP) [2], [6], [7], to optimize large polynomial multiplications from $O(N^2)$ to $O(N \log N)$. More specifically, let $A(x)$ and $B(x)$ be two polynomials (on the ring of $\mathbb{Z}_p[x]/\Phi(x)$ where $\Phi(x)$ is usually the degree-N cyclotomic polynomial). Their product $C(x) = A(x) \cdot B(x)$ could be obtained as the following:

$$C = \mathsf{iNTT}(\mathsf{NTT}(A) \odot \mathsf{NTT}(B))$$

where $\odot$ denotes an element-wise vector multiplication.

However, the specific values of the parameters N and W in NTT vary significantly across different algorithms, and even among applications of the same algorithm. For state-of-the-art HE schemes like BGV [3] and CKKS [5], N is typically set to a moderately large power-of-2 value that is at least 2^{16}, in order to support 128-bit security and achieve a balance between sufficient multiplicative depths and manageable bootstrapping cost [19], [28], [29]. In addition, HE schemes are usually able to choose a "good" finite field $\mathbb{Z}_p$ to make p the product of some smaller co-prime numbers. Each element in the original large field can be decomposed into a unique set of elements in the fields of these smaller moduli, forming a Residue Number System (RNS). Polynomial additions and multiplications in the large field can then be performed in RNS, independently on each of the decomposed polynomials with the small moduli. Therefore, NTT and iNTT computations in HE typically work with W around 50 to 60 bits, within the native integer type width of modern processors (64 bits).

On the other hand, ZKP applications would usually require much larger NTT and iNTT operations, with N reaching 2^{21} to 2^{27} in real-world protocols like Zcash [26] and Filecoin [27]. They also need to use large finite fields, but modulo an exact power-of-2 value that cannot be factorized in the RNS way. Therefore, NTT and iNTT in ZKP directly handle wide integer data with hundreds of bits, e.g., 256, 381, and 753 bits for the BN128, BLS12-381, and MNT4-753 elliptic curves commonly used in various ZKP implementations [30]–[33].

C. Related Work

There have been many prior efforts focusing on accelerating NTT using FPGAs. Roy et al. [8] proposed an FPGA-based NTT accelerator that supported $N = 2^{15}$ and $W = 30$ bits. They supported strided data accesses by merging two required coefficients in a single memory word. Later, they also implemented an NTT co-processor [9] that used two parallel butterfly units, but limited to $N = 2^{12}$. HEAX [10] realized a fully pipelined architecture with multiple NTT cores for the CKKS HE algorithm. With larger FPGA chips, HEAX could scale between $N = 2^{12}$ and 2^{14}, and worked with $W = 54$ bits. Other FPGA-based proposals [11]–[14] were also designed and synthesized for relatively small and fixed NTT parameters. When scaling them up to accommodate larger NTT kernels, the limited on-chip resources and off-chip bandwidth would restrict the performance. To our best knowledge, CycloneNTT [15] was the first FPGA accelerator specifically designed to target large NTT sizes. It used a simple butterfly network as the core computing unit, and adopted the special Goldilocks field [34] to minimize the resource usage. But the fixed depth of the butterfly network d made it less flexible, only supporting $N = 2^{kd}, k \in \mathbb{Z}$. Furthermore, it did not use on-the-fly twiddle generation as we do.

Recent HE and ZKP accelerators were mostly implemented as ASIC. They mainly used the multi-dimensional decomposition of NTT (Section II-D) to reduce the required hardware resources. PipeZK [16] ($N = 2^{20}$, $W = 256$ bits) and F1 [17] ($N = 2^{14}$, $W = 32$ bits) both used 2-D NTT decomposition. CraterLake [18] ($N = 2^{16}$, $W = 28$ bits) handled NTT similarly to F1, but with more NTT units and an optimized transpose network. BTS [19] went one step further and performed 3-D decomposition, breaking a 2^{17}-size NTT into $2^6 \times 2^5 \times 2^6$. It used 2048 on-chip processing elements, each capable of executing a 2^6-size NTT. Despite great performance, these ASIC designs only considered specific parameter choices of N and W in one type of algorithm, and could not adapt to diverse applications.

On GPUs, Govindaraju et al. [20] proposed FFT/DFT optimizations suited to different problem sizes. They decomposed a large-size FFT into smaller kernels that fit in the limited shared memory. Kim et al. [21] ported various GPU optimizations that originally targeted DFT to accelerate NTT. They proposed dynamic generation of twiddle factors to reduce off-chip bandwidth. Özerk et al. [22] proposed a hybrid approach that reduced data dependencies and created more parallelism opportunities, to best exploit the abundant GPU resources.

D. Multi-Dimensional Decomposition for NTT

Previous NTT hardware designs [16]–[18] have adopted the 2-D NTT decomposition method that converts a large NTT of size N into multiple smaller kernels of size I and size J, satisfying $N = I \times J$. Generally, this 2-D NTT decomposition can be broken down into three distinct steps.

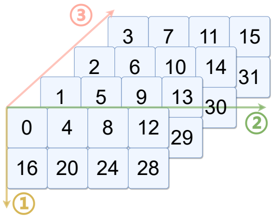


Fig. 1. Multi-dimensional NTT decomposition example.

1) Organize the input data of N elements into a 2-D plane with I columns and J rows in a row-major order. Then, perform size-J NTT on each of the I columns.
2) Multiply each element at the coordinate (r, c) on the 2-D plane with the corresponding twiddle factor ω_N^{rc}.
3) Perform size-I NTT on each of the J rows. Then, transform the 2-D plane following a column-major order, back into a 1-D sequence of N elements as the output.

In this 2-D decomposition, setting $I = J = \sqrt{N}$ reaches the best utilization, where we only need fewer hardware resources to process smaller NTT kernels of size $O(\sqrt{N})$ rather than $O(N)$, reducing the chip area requirement.

We further generalize to *multi-dimensional NTT decomposition* [20]. In fact, the NTT operations on each decomposed dimension can be regarded as standalone NTT kernels, and thus decomposed again, expanding the total number of dimensions in a hierarchical and recursive manner. Mathematically, we decompose a size-N NTT into a d-dimensional hypercube, i.e., $N = m \times n^{d-1}$, where $m \leq n$ represents an incomplete dimension to support more general values of N. The remaining dimensions all have the same size n. Our work assumes N, n, and m are all power-of-2 numbers, so having one incomplete dimension supports any power-of-2 values of N. In Fig. 1, $N = 2^5, n = 2^2, m = 2^1, d = 3$. With such a decomposition, we need to perform n^{d-1} times of size-m NTT (① in Fig. 1) and $m \times n^{d-2}$ times of size-n NTT (② and ③) along each dimension in turn. Similar to the 2-D case, twiddle factor multiplications and dimension transpositions are required in between of these NTTs whenever we switch the dimensions.

III. SAM Design

In this work, we propose SAM, an FPGA-based, scalable NTT accelerator. SAM leverages multi-dimensional decomposition to efficiently process NTT kernels of various sizes from 2^{16} up to 2^{28} for state-of-the-art HE and ZKP schemes. This section presents the hardware design of SAM.

A. Architecture Overview

Fig. 2 illustrates the overall system architecture of SAM, which is implemented on an FPGA connected to the host CPU via a PCIe bus. The host CPU transfers the input data to the DDR memory on the FPGA board, and sends the commands

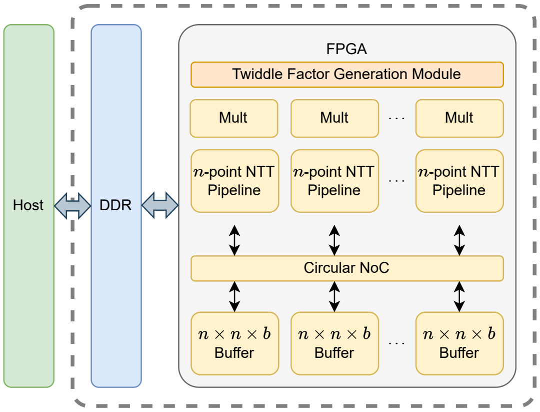

FPGA
Twiddle Factor Generation Module
Mult
Mult
Mult
n-point NTT Pipeline
n-point NTT Pipeline
n-point NTT Pipeline
Circular NoC
$n \times n \times b$ Buffer
$n \times n \times b$ Buffer
$n \times n \times b$ Buffer
Host
DDR

Fig. 2. SAM architecture overview.

and the input arguments to the FPGA engine to start processing. Since we target any large-size NTT processing, we assume both the input data and the constant twiddle factors cannot fit on-chip and are accessed from the off-chip DDR memory.

At the center of the SAM architecture are multiple pipelined NTT processing units that can natively execute size-n or smaller NTT operations. Each NTT pipeline is accompanied with an extra modular multiplier, which performs the twiddle factor multiplications when switching across different decomposed dimensions (Section II-D). The input and output data elements of the NTT pipelines are buffered on-chip, where each buffer holds $n \times n \times b$ elements with double buffering, i.e., in total $2n^2bW$ bits. We process two decomposed dimensions in succession for each round of data fetch (thus n^2) to improve on-chip data reuse. $b \geq 1$ is a buffer capacity extension factor that allows us to prefetch more data to better utilize DDR bandwidth and hide NTT pipeline delays (Section III-E). We define a compute *lane* to include one NTT pipeline, one modular multiplier, and one buffer. We use t to denote the number of lanes instantiated in SAM. Section IV discusses how to concretely set the above design parameters, n, t, b.

With the fixed hardware that natively supports size-n NTT processing, we can map an NTT kernel of arbitrary power-of-2 size N onto SAM, by decomposing it into $d = \lceil \log_n N \rceil$ dimensions, where $N = m \times n^{d-1}$ for some $m \leq n$. A larger N will result in a larger d and needs proportionally longer processing time on the same amount of resources. Section III-B describes the generic computation flow and the control logic in SAM that supports any value of d. Because multi-dimensional NTT decomposition also complicates the usage of twiddle factors and the strided butterfly data access patterns, SAM also incorporates several novel optimizations, including a specialized on-the-fly twiddle factor generation scheme to save off-chip memory bandwidth (Section III-C), and an efficient on-chip data layout scheme that avoids buffer access conflicts (Section III-D).

B. Computation Flow

In each compute lane of SAM, the NTT pipeline natively processes a size-n NTT operation, while the buffer has a capacity to hold 2-D planes of size $n \times n$. This is because on most FPGA chips, the BRAM/URAM resources (for on-chip buffers) are relatively abundant compared to the DSP resources (for modular multipliers in NTT pipelines). Increasing the buffer capacity makes better use of the RAM resources to improve on-chip data reuse. We empirically find 2-D fusion is a good choice. We fetch data from a 2-D plane from the off-chip memory into the buffer of one lane, and apply both the row and column NTT operations on the data to save 2× off-chip data traffic. We call such 2-D processing of two decomposed dimensions as one *round*.

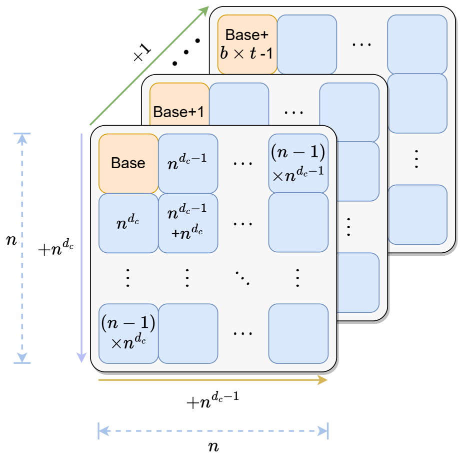
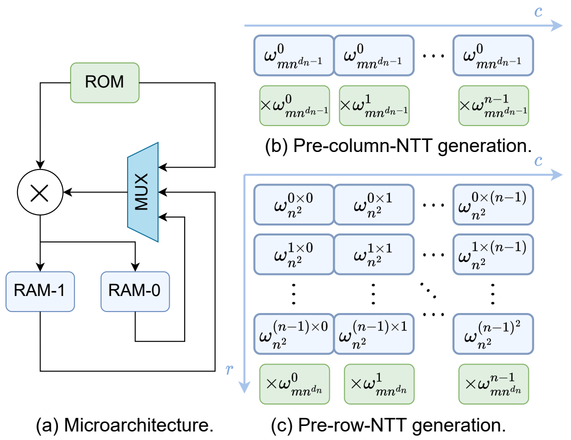


Fig. 3. The $b \times t$ 2-D data planes buffered and processed by the t compute lanes. The elements along the row and column dimensions have strides of n^{d_c-1} and n^{d_c}, respectively, and are likely non-consecutive for most of the time. The exact layout in the on-chip buffers is depicted in Fig. 5.

We now describe the computation flow of SAM for an arbitrary N and $d = \lceil \log_n N \rceil$. The control logic maintains the current processing dimension index d_c, which is initialized to $d-1$ by the input command from the host CPU. The hardware then starts a round of processing on the 2-D planes of the dimensions d_c and $d_c - 1$, including the row and column NTT operations and the twiddle factor multiplications. After the round finishes, it decrements d_c by 2. The hardware repeats with more rounds, until d_c gets to 1. If d is not even, before the above sequence of normal rounds, we first do a special round, which only applies 1-D NTT operations on the d_c dimension, and decrements d_c by 1. In both cases, the incomplete dimension m (if any) is processed as the first one.

In one round for the dimensions d_c and $d_c - 1$, there are n^{d-2} (when $d_c = d-1$) or $n^{d-3} \times m$ (when $d_c < d-1$) 2-D planes in total. Each time t planes are processed in the t lanes in parallel. We make each group of these t planes have consecutive base addresses, as shown in Fig. 3. The base address of the first plane starts from 0, and increments by t each time, until reaching n^{d_c-1}, at which point it is set to n^{d_c+1} (when $d_c < d-1$), and repeats. Such an order allows the 2-D data planes that share the same set of twiddle factors to be processed together, enabling on-the-fly twiddle factor generation in Section III-C.


ROM
MUX
RAM-1
RAM-0
c
$\omega^0_{mn^{d_n-1}}$
$\times\omega^0_{mn^{d_n-1}}$
$\times\omega^1_{mn^{d_n-1}}$
$\times\omega^{n-1}_{mn^{d_n-1}}$
c
r
$\omega^{0\times0}_{n^2}$
$\omega^{0\times1}_{n^2}$
$\omega^{0\times(n-1)}_{n^2}$
$\omega^{1\times0}_{n^2}$
$\omega^{1\times1}_{n^2}$
$\omega^{1\times(n-1)}_{n^2}$
$\omega^{(n-1)\times0}_{n^2}$
$\omega^{(n-1)\times1}_{n^2}$
$\omega^{(n-1)^2}_{n^2}$
$\times\omega^0_{mn^{d_n}}$
$\times\omega^1_{mn^{d_n}}$
$\times\omega^{n-1}_{mn^{d_n}}$

(a) Microarchitecture.

(b) Pre-column-NTT generation.

(c) Pre-row-NTT generation.

Fig. 4. On-the-fly twiddle factor generation scheme in SAM.

C. On-the-Fly Twiddle Factor Generation

In multi-dimensional decomposition, there are two places that need twiddle factors. First, each NTT pipeline needs its local twiddle factors to complete the size-n NTT. The number of these constants is small and bounded by the hardware design parameters, only $O(tn)$, so they are directly stored in the ROM of the pipeline stages. Second, when we switch dimensions in the decomposed hypercube, we need to do twiddle factor multiplications in between. Such twiddle factors are $O(N)$, corresponding to the workload size and could be very large. Thus off-chip DDR accesses are necessary for them.

We use an on-the-fly twiddle factor generation module to reduce the off-chip traffic demands. Compared to previous work that only implemented on-the-fly generation for fixed and limited decomposed depths [12], [19], [21], SAM generalizes the scheme and is able to support arbitrary decomposition. Fig. 4(a) shows the microarchitecture, including a modular multiplier, a ROM, and two output double-buffered RAMs. The ROM stores a small amount of seed twiddle factors, which are used by the multiplier at runtime to iteratively calculate the needed twiddle factors, and store to the output RAMs.

As we buffer 2-D data planes on-chip in each round, the generation of twiddle factors has two types.

- **Pre-column-NTT generation.** Before doing the column NTTs for dimension d_c, all columns of the 2-D data plane should be multiplied with the same set of twiddle factors. We use a RAM of n elements to store these factors, and iteratively update them after processing every n^{d_c-1} planes of size mn (when $d_c = d-1$) or $n^{d_c-2} \times m$ planes of size n^2 (when $d_c < d-1$). Specifically, let $d_n = d - d_c$. As in Fig. 4(b), we initialize all the factors to $\omega^0_{mn^{d_n-1}} = 1$ (or N^{-1} mod p for iNTT in the first round), which are used by the first group of planes. The next set of factors are obtained by multiplying $\omega^r_{mn^{d_n-1}}$ (read from the ROM) at row r to each current factor.
- **Pre-row-NTT generation.** The case is more complicated for the row NTTs for dimension $d_c - 1$. Each element of the 2-D plane should be multiplied with a different twiddle factor. We need double-buffered RAMs of size n^2, as shown in Fig. 4(c). When changing to the next set (the same pace as in pre-column-NTT generation), we need to multiply each factor at column c with $\omega^c_{mn^{d_n}}$.

In total, we need $2(n+n^2)W$ bits of RAM, e.g., up to 17 kB for $n = 16$ and $W = 256$ bits, independent of N. After a round is completed, the module is reset and the twiddle factors of the next round will be generated from scratch.

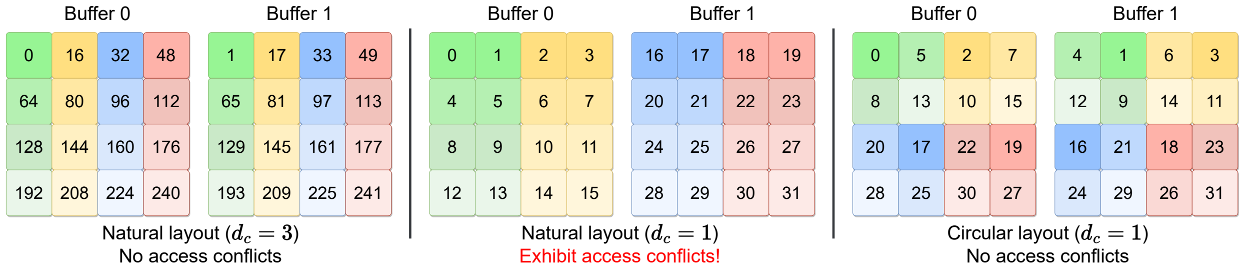
We also need a set of $\omega^i_{n^j}$, for $i = 0, 1, \ldots, n-1$, and $j = 2, 3, \ldots, d_{\max}$, to update the generated twiddle factors. In addition, the row twiddle factor generation needs n^2 initial values. All these constants consume $n(d_{\max}-1)W + n^2W$ bits of ROM. This ROM capacity *does* depend on the maximum supported NTT size $N_{\max} = n^{d_{\max}}$, but only increases linearly with $d_{\max} = \log_n N_{\max}$. This is the only resource in SAM that depends on N. In reality, this logarithmic increasing is not an issue. For example, assuming $n = 16$, $W = 256$ bits, $N_{\max} = 2^{28}$, the ROM is only 11 kB. Even extending $N_{\max}$ to a huge size of 2^{36} only increases the ROM capacity by 1 kB.

D. Data Layout in On-Chip Buffers

As discussed in Section III-B, each time SAM processes t data planes of $n \times n$ elements in the decomposed hypercube. When $d_c > 1$, the addresses of these elements are consecutive only along the plane dimension, but have large strides along the rows and columns inside each plane (Fig. 3). Therefore, each off-chip DDR access brings in the t elements at the same position on all the planes, and they can be independently stored into the t separate buffers in different lanes, without buffer conflicts. This ensures that the data needed by each NTT pipeline are kept local in each lane. Fig. 5 left shows such an example. The number in each element denotes its address, and two elements with the same color (with consecutive addresses) are read from memory and stored to the buffers in the same cycle. The layout of the two buffers matches exactly with the two planes needed by the NTT pipelines as in Fig. 3.

However, the above natural layout does not work for the last round when d_c becomes 1. This case is special as the data within each plane become consecutive, e.g., the first row should have $0, n^{d_c-1} = 1, \ldots$. Therefore, consecutive elements 0 and 1 fetched together from the memory now need to be put in the same buffer (Fig. 5 middle), causing access conflicts. In SAM we require $n > t$, so only $d_c = 1$ has this issue.

To resolve this issue, we propose a novel circular layout shown on Fig. 5 right. The elements needed by the t lanes are circularly distributed across all the buffers after being fetched from off-chip. Specifically, a perfect layout should satisfy two conditions. First, the t consecutive elements fetched together (with the same color) must be stored to different buffers to avoid conflicts. Second, the on-the-fly twiddle factor generation order when $d_c = 1$ in Section III-C requires us to use the t lanes for multiple row/column NTT operations on one 2-D plane, because now only $n^{d_c-1} = 1$ plane uses the same set of twiddle factors. Therefore, each group of t consecutive elements along the rows and columns of the same plane (e.g., 0 and 1, 0 and 4, 16 and 17, 16 and 20) should also be stored



Buffer 0				Buffer 1			
0	16	32	48	1	17	33	49
64	80	96	112	65	81	97	113
128	144	160	176	129	145	161	177
192	208	224	240	193	209	225	241

Natural layout ($d_c = 3$)
No access conflicts

Buffer 0				Buffer 1			
0	1	2	3	16	17	18	19
4	5	6	7	20	21	22	23
8	9	10	11	24	25	26	27
12	13	14	15	28	29	30	31

Natural layout ($d_c = 1$)
Exhibit access conflicts!

Buffer 0				Buffer 1			
0	5	2	7	4	1	6	3
8	13	10	15	12	9	14	11
20	17	22	19	16	21	18	23
28	25	30	27	24	29	26	31

Circular layout ($d_c = 1$)
No access conflicts

Fig. 5. On-chip buffer layout schemes in SAM, with $t = 2$ and $n = 4$. The numbers in the blocks denote the global element indices. Each two elements with the same color are fetched from the off-chip DDR together in the same cycle. Left: The natural layout works well with no buffer conflicts when $d_c > 1$. Middle: The natural layout incurs buffer conflicts when $d_c = 1$. Two elements fetched in the same cycle need to be stored to the same buffer. Right: The circular layout resolves all buffer conflicts for $d_c = 1$, by circulating and distributing elements across all buffers.

in different buffers, so that we can read out one element from each of the t rows/columns in parallel for the t NTT pipelines.

Our circular layout scheme satisfies the above requirements in the following way. For an element at the position of row r, column c, and plane p (all starting from 0) in the natural layout, e.g., 24 is at position $(r = 2, c = 0, p = 1)$, we put it to the position (r', c', p') in the circular layout, where

$$k = (p + r) \bmod t \tag{1}$$
$$r' = (p \cdot n + r)/t \tag{2}$$
$$c' = c \tag{3}$$
$$p' = (c + k) \bmod t \tag{4}$$

For example, 24 with $(r = 2, c = 0, p = 1)$ has $k = 1$, and is put to $(r' = 3, c' = 0, p' = 1)$. The idea is to distribute the t elements (with consecutive column indices c) to the t buffers (with consecutive p') as in Equation (4), following a certain circular shift distance k. The shift distance k is determined by the original row index r in Equation (1), so elements in consecutive rows are also distributed to different buffers.

We use a circular shift network-on-chip (NoC) across the multiple compute lanes, to realize the mapping from c to p' for the t elements in each DDR access. It calculates the required k value for shifting the elements. Since k can only have t possible values (usually 4 to 16 in our design), we use pipelined multiplexers to implement the shift network in a straightforward way. We implement a simple buffer address generator at each buffer to calculate r' and c', which determines the location in the buffer to write the data to.

E. Additional Hardware Optimizations

NTT pipelines. SAM adopts its NTT pipeline design from PipeZK [16]. As a size-n NTT unit, the pipeline has $\log n$ stages. There are input/output FIFO buffers before/after each stage, whose FIFO depth is equal to the desired access stride, i.e., $\frac{n}{2^i}$ for stage i. Each stage of the pipeline receives one input element per cycle, and outputs one result per cycle. The FIFO buffers organize the data to realize the expected strided accesses. This design has several advantages that match our multi-dimensional decomposition. First, the pipeline reads and writes one element per cycle, which reduces the data bandwidth requirement and is very friendly to non-consecutive data accesses. Second, it is easy to support the incomplete dimension (if any) $m < n$ for general power-of-2 values of N, by skipping the first few stages in the pipeline. Third, this design has good scalability on FPGA, especially for the way to support strided data selection, where FIFOs are relatively easy to instantiate than a large number of multiplexers [10].

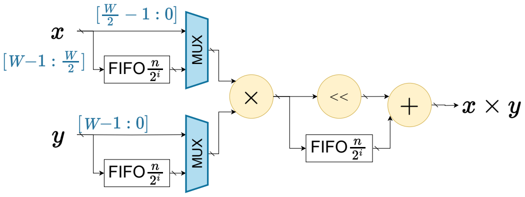


Fig. 6. Using a $\frac{W}{2}$-bit $\times$ W-bit modular multiplier to perform W-bit $\times$ W-bit multiplications with half of the throughput.

However, this NTT pipeline has a critical issue, where the fully pipelined modular multiplier at each stage is only 50% utilized. This is because only one element enters the stage in each cycle, but it requires two elements to compute. The stage i works for $\frac{n}{2^i}$ cycles and then sits idle for another $\frac{n}{2^i}$ cycles.

To tackle this issue, we propose an improved design that uses a half-throughput modular multiplier at each pipeline stage. We reduce the modular multiplier width from W-bit $\times$ W-bit to $\frac{W}{2}$-bit $\times$ W-bit, and add additional operand and result FIFOs before and after it, as shown in Fig. 6. Specifically, for input operands x and y, we first compute $x_L \times y$ (we omit modulo in this paragraph) and store to the result FIFO, while buffering both x_H and y in the operand FIFOs. After all input data of this phase are fed in and during the idle time, x_H and y are popped from the operand FIFOs for computing, and the result is shifted and added to the corresponding product $x_L \times y$ from the result FIFO, to obtain the final result $x \times y$. In many cases y is always a constant, and the second operand FIFO can be removed. This optimization reduces the DSP consumption by roughly 1/3 without any negative performance impact.

Data prefetching. So far, SAM uses t compute lanes to process t 2-D data planes in parallel. In general, these 2-D planes have non-consecutive elements along the rows and columns, while Section III-B ensures that the elements along the plane dimension are always consecutive, and thus fetching multiple planes together increases the DDR burst access granularity and correspondingly the bandwidth utilization. However, t is limited by the available FPGA logic resources. On a typical FPGA, DSP units are more precious than RAM blocks. If we only fetch and buffer t planes, we would underutilize the RAM space, while the DDR access burst length is still insufficient.

We hence introduce a buffer capacity extension factor b to decouple data processing and data buffering. SAM actually fetches b groups of t 2-D planes together, with the buffer in each lane also enlarged by $b\times$. These b groups of planes are processed sequentially. All the $b \times t$ planes within one fetch have consecutive base addresses, making each DDR access of btW bits. Thus a sufficiently large b can fully utilize the on-chip RAM space to achieve high DDR bandwidth utilization.

Having $b\times$ more available work on-chip also helps hide the long NTT pipeline latency. When switching from column NTTs to row NTTs, we must wait until the output from the last column NTT to be written back to the buffer, before we can read out the first row to start the first row NTT. In our multi-stage NTT pipeline [16], such dependencies result in long pipeline stalls. With b groups of planes, we adjust the execution order, to first perform the row NTTs on all b groups, and then start the column NTTs.

IV. Design Space Exploration

SAM contains three main design parameters, the size of each NTT pipeline n, the number of parallel lanes t, and the buffer capacity extension factor b. They exhibit various tradeoffs. A larger n consumes more on-chip resources, but also supports larger NTT kernels natively and reduces the decomposition overheads. Increasing t improves parallelization and thus performance, but is limited by both the on-chip DSP resources and the off-chip DDR bandwidth. Finally, b should be large enough to better utilize the memory bandwidth, subject to the FPGA RAM space limit.

To determine these parameters, we consider latency matching between computations and data accesses for double buffering. SAM uses one DDR channel for data reads and another for writes. We empirically find on our platform that the DDR burst length per channel must be no less than $L = 1\,\mathrm{kB}$ to achieve the maximum sustainable bandwidth of $B = 11\,\mathrm{GB/s}$. When $W = 256$ bits, we can achieve a frequency of $f = 100\,\mathrm{MHz}$. For the compute latency, there are $\frac{N \log N}{2}$ butterfly operations, and our NTT pipeline needs 2 cycles for each. So

$$\mathrm{Latency}_{\mathrm{comp}} = \frac{N}{tf \log m} + \frac{N(\log N - 1)}{tf \log n}$$

where the first term specially considers the incomplete dimension m. For the memory latency, we multiply the data size NW by the number of rounds, and divide by the bandwidth.

$$\mathrm{Latency}_{\mathrm{mem}} = \frac{NW}{B} \left\lceil \frac{\log N}{2 \log n} \right\rceil$$

Letting the two latencies equal, and simplifying by omitting the ceiling function and assuming $m = n$, we get $t = \frac{2B}{Wf} \approx 6.9$. Because t must be a power of 2 as required by the buffer layout in Section III-D, it may be 4 or 8. We find that for large N values, there are many large strided accesses that decrease the effective bandwidth B, which in turn lowers the optimal value of t. We thus choose $t = 4$. Accordingly, $b = \frac{L}{Wt} = 8$.

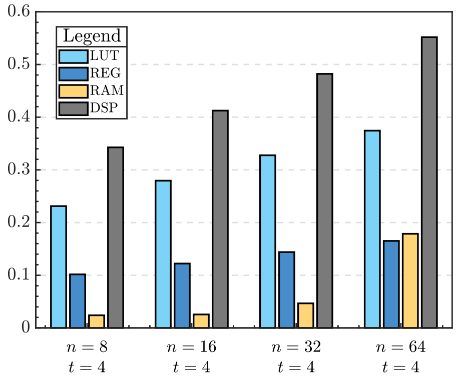


Fig. 7. Resource consumption of various n values for $t = 4$ at 256 bits.

Finally, to decide n, we see both the compute and memory access latencies above decrease with $\log n$, making us use an n as large as possible. However, we are limited by the on-chip FPGA resources. As shown in Fig. 7, the DSP count eventually restricts SAM to use $n = 64$.

Similarly, we calculate another configuration of SAM for $W = 64$ bits with a frequency $f = 165\,\mathrm{MHz}$. Repeating the above procedure, we have $t = 16$, $b = 8$, $n = 64$. Narrower multipliers use fewer DSPs and allow for more parallel lanes.

V. Evaluation

In this section, we evaluate SAM against the CPU baseline and previous NTT accelerators on FPGAs.

A. Experimental Methodology

We implement the RTL design of SAM in Verilog, and synthesize it using Xilinx Vitis 2021.1, targeting the Alveo U250 card with 12284 DSP slices, 2547 instances of 36 kbit BRAM, and 1280 instances of 288 kbit URAM. We set the target frequency to 100 MHz (256 bits) or 165 MHz (64 bits) for the core logic, and 300 MHz for the I/O and memory interface. We have extensively verified the correctness using random tests and comparing with the CPU baseline [30]. The host server has an Intel Xeon Gold 5120 CPU with 14 physical cores at 2.2 GHz, and 252 GB of DDR4 memory.

We compare SAM with several state-of-the-art FPGA-based NTT accelerators [8], [12], [14]–[16], as well as a CPU baseline that uses libsnark [30] optimized for large-scale 256-bit NTTs in ZKP. We evaluate NTT sizes in the range of 2^{16} to 2^{28}. This range is commonly used for HE and ZKP applications. It not only covers the problem sizes in previous work [8]–[13], [17]–[19], but also extends to larger scales rarely supported before. For each bitwidth W, the same hardware configuration can work for different N values. All reported results of SAM are from real on-board executions. The input data of the desired size N are initially stored in

TABLE I
OVERALL COMPARISON RESULTS.

Design	**Platform**	W **(bits)**	**Freq** (MHz)	**Data**	**LUT / REG / RAM** (kb) **/ DSP**	N	**Latency** (ms)
SAM (**Ours**)	XCU250	256	100 & 300	Off-chip	593073 / 533675 / 81684 / 6776	2^{16} 2^{20} 2^{24} 2^{28}	1.24 12.61 183.56 4023.49
CPU [30]	Intel Xeon	256	2200	Off-chip	-	2^{16} 2^{20} 2^{24}	511.47 1244.68 19970.75
PROTEUS [14]	XCU250	256	125	***On-chip***	356000 / - / 55620 / 2640	2^{16}	1.05
PipeZK-Scaled [16]	-	256	100 (scaled)	Off-chip	-	2^{20}	33.00
SAM (**Ours**)	XCU250	64	165 & 300	Off-chip	267132 / 328486 / 76536 / 2736	2^{16} 2^{17} 2^{18} 2^{20} 2^{24} 2^{28}	0.38 0.56 0.99 2.84 34.12 750.14
HEPCloud [8]	XC6VLX240T	30	100 & 200	***On-chip***	72163 / 63086 / 3420 / 250	2^{16}	0.48
FCCM'20 [12]	XCVU190	62	200	***On-chip***	365000 / - / 81304 / 1332	2^{17}	0.98
PROTEUS [14]	XCVX485T	64	135	***On-chip***	31300 / - / 9180 / 300	2^{16}	0.44
CNTT-4 [15]	VU55P	64	300	***HBM***	53026 / 46026 / 13824 / 74	2^{20} 2^{24}	2.12 42.57
CNTT-6 [15]	VU55P	64	161	***HBM***	563677 / 319104 / 82944 / 691	2^{18} 2^{24}	0.10 8.08
CNTT-6-Scaled [15]	-	64	161	Off-chip	-	2^{18} 2^{24}	0.88 71.44

the FPGA board memory. The latency measurements do not include data transfers between the host and the FPGA board.

B. Results

Table I summarizes the overall comparison between SAM and the baselines. First, for $W = 256$ bits, SAM uses $t = 4$, $b = 8$, $n = 64$. It achieves $376\times, 99\times, 109\times$ speedups at $N = 2^{16}, 2^{20}, 2^{24}$, respectively, compared to the CPU baseline. Among previous FPGA-based NTT accelerators, only PROTEUS [14] reported performance for 256 bits, but it assumed all data fit on-chip and only supported up to $N = 2^{16}$. Nevertheless, SAM almost matches this on-chip performance with 18% slowdown despite that its data are from off-chip, demonstrating that our design parameter selection in Section IV effectively hides most memory access latencies. We also compare with a scaled version of PipeZK [16]. PipeZK was an ASIC design and had a much higher frequency. We scale down the frequency to be the same as SAM, and optimistically assume linear performance scaling. SAM outperforms PipeZK-Scaled by 2.6× at $N = 2^{20}$, the largest size PipeZK could support.

For the 64-bit version of SAM, we use $t = 16$, $b = 8$, $n = 64$. HEPCloud [8], FCCM'20 [12], and PROTEUS [14] were all on-chip designs and only supported small NTT sizes up to 2^{16} or 2^{17}. Nevertheless, SAM is able to perform slightly faster than all of them, even though it needs to access off-chip data. CNTT-4 and CNTT-6 were two different configurations in CycloneNTT [15], which used an HBM-capable FPGA board. CNTT-4 utilized 6 HBM channels, while CNTT-6 utilized 24 HBM channels; but they only supported $\log N$ as multiples of 4 and 6, with limited flexibility. It is not surprising the much higher memory bandwidth helps CycloneNTT perform better. However, if we scale down their bandwidth to match SAM, e.g., by 0.11× from CNTT-6, SAM would achieve a 2.1× speedup at $N = 2^{24}$. Furthermore, CycloneNTT was specially optimized for the Goldilocks field, which reduced the resource consumption and boost its frequency. SAM could also be specialized to a certain field, but we opt for generality.

VI. CONCLUSION

In this paper, an FPGA-accelerated NTT architecture, SAM, is designed based on the recursive multi-dimensional decomposition of the NTT algorithm. SAM can flexibly support a wide range of NTT kernel sizes commonly seen in HE and ZKP algorithms, up to 2^{28}. It has specific optimizations for its on-chip data layout and off-chip data accesses under the execution flow of multi-dimensional decomposition. SAM achieves significant speedups over previous NTT accelerators.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262).

References

[1] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 2009, p. 169–178.

[2] J. Groth, "On the Size of Pairing-Based Non-interactive Arguments," in *Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016, pp. 305–326.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," *ACM Transactions on Computation Theory*, vol. 6, no. 3, Jul 2014.

[4] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," Cryptology ePrint Archive, 2012. [Online]. Available: https://eprint.iacr.org/2012/144

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Proceedings of the 23rd International Conference on the Theory and Applications of Cryptology and Information Security*, 2017, pp. 409–437.

[6] J. Groth and M. Maller, "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs," in *Annual International Cryptology Conference*, 2017, pp. 581–612.

[7] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A Distributed Zero Knowledge Proof System," Cryptology ePrint Archive, 2018. [Online]. Available: https://eprint.iacr.org/2018/691

[8] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-Based Multicore Processor for FV Somewhat Homomorphic Function Evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.

[9] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[10] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 1295–1309.

[11] N. Zhang, Q. Qin, H. Yuan, C. Zhou, S. Yin, S. Wei, and L. Liu, "NTTU: An Area-Efficient Low-Power NTT-Uncoupled Architecture for NTT-Based Multiplication," *IEEE Transactions on Computers*, vol. 69, no. 4, pp. 520–533, 2020.

[12] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," in *Proceedings of the 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 56–64.

[13] A. C. Mert, E. Karabulut, E. Öztürk, E. Savaş, M. Becchi, and A. Aysu, "A Flexible and Scalable NTT Hardware: Applications from Homomorphically Encrypted Deep Learning to Post-Quantum Cryptography," in *Proceedings of the 23rd Conference on Design, Automation and Test in Europe (DATE)*, 2020, pp. 346–351.

[14] F. Hirner, A. C. Mert, and S. S. Roy, "PROTEUS: A Tool to Generate Pipelined Number Theoretic Transform Architectures for FHE and ZKP Applications," Cryptology ePrint Archive, 2023. [Online]. Available: https://eprint.iacr.org/2023/267

[15] K. Aasaraai, E. Cesena, R. Maganti, N. Stalder, J. Varela, and K. Bowers, "CycloneNTT: An NTT/FFT Architecture Using Quasi-Streaming of Large Datasets on DDR- and HBM-based FPGA Platforms," Cryptology ePrint Archive, 2022. [Online]. Available: https://eprint.iacr.org/2022/1657

[16] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 416–428.

[17] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, p. 238–252.

[18] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, p. 173–187.

[19] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, p. 711–725.

[20] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC)*, 2008, pp. 1–12.

[21] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs," in *Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 264–275.

[22] O. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption," *Journal of Supercomputing*, vol. 78, no. 2, p. 2840–2872, Feb 2022.

[23] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes," in *Selected Areas in Cryptography (SAC)*, 2017, pp. 423–442.

[24] S. Halevi, Y. Polyakov, and V. Shoup, "An Improved RNS Variant of the BFV Homomorphic Encryption Scheme," in *Topics in Cryptology (CT-RSA)*, 2019, pp. 83–105.

[25] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," in *Selected Areas in Cryptography (SAC)*, 2019, pp. 347–368.

[26] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 459–474.

[27] "Filecoin: A Decentralised Market for Storage," https://filecoin.io/filecoin.pdf, 2022.

[28] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.

[29] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption," in *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 882–895.

[30] "libsnark: a C++ library for zkSNARK proofs," https://github.com/scipr-lab/libsnark, 2022.

[31] "bellperson: GPU parallel acceleration for zk-snark," https://github.com/filecoin-project/bellperson, 2022.

[32] "bellman: a crate for building zk-snark circuits," https://github.com/zkcrypto/bellman, 2022.

[33] "jsnark: A java library for building snarks," https://github.com/akosba/jsnark, 2022.

[34] PolygonZero, "Plonky2: Fast Recursive Arguments with PLONK and FRI." https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf, 2022.