2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071017

TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU

Shengyu Fan*‡, Zhiwei Wang*†, Weizhi Xu‡, Rui Hou*†, Dan Meng*†, Mingzhe Zhang*
* State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China.
† School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.
‡ School of Information Science and Engineering, Shandong Normal University, Jinan, China.
damionfan@163.com, {wangzhiwei, hourui, mengdan, zhangmingzhe}@iie.ac.cn, xuweizhi@sdnu.edu.cn

***Abstract*—In the cloud computing era, privacy protection is becoming pervasive in a broad range of applications (e.g., machine learning, data mining, etc). Fully Homomorphic Encryption (FHE) is considered the perfect solution as it enables privacy-preserved computation on untrusted servers. Unfortunately, the prohibitive performance overhead blocks the wide adoption of FHE (about $10,000\times$ slower than the normal computation). As heterogeneous architectures have gained remarkable success in several fields, achieving high performance for FHE with specifically designed accelerators seems to be a natural choice. Until now, most FHE accelerators have focused on efficiently implementing one FHE operation at a time based on ASIC and with significantly higher performance than GPU and FPGA. However, recent state-of-the-art FHE accelerators rely on an expensive and large on-chip storage and a high-end manufacturing process (i.e., 7nm), which increase the cost of FHE adoption.**

In this paper, we propose TensorFHE, an FHE acceleration solution based on GPGPU for real applications on encrypted data. TensorFHE utilizes Tensor Core Units (TCUs) to boost the computation of Number Theoretic Transform (NTT), which is the part of FHE with highest time-cost. Moreover, TensorFHE focuses on performing as many FHE operations as possible in a certain time period rather than reducing the latency of one operation. Based on such an idea, TensorFHE introduces operation-level batching to fully utilize the data parallelism in GPGPU. We experimentally prove that it is possible to achieve comparable performance with GPGPU as with state-of-the-art ASIC accelerators. TensorFHE performs 913 KOPS and 88 KOPS for NTT and HMULT (key FHE kernels) within NVIDIA A100 GPGPU, which is 2.61$\times$ faster than state-of-the-art FHE implementation on GPGPU; Moreover, TensorFHE provides comparable performance to the ASIC FHE accelerators, which makes it even 2.9$\times$ faster than the F1+ with a specific workload. Such a pure software acceleration based on commercial hardware with high performance can open up usage of state-of-the-art FHE algorithms for a broad set of applications in real systems.

I. Introduction

As cloud computing servers are becoming the infrastructure of the world, their security is gaining increasing attention. However, conventional techniques can hardly mitigate all kinds of attacks since defense technology always lags behind new attacking methods. Fortunately, *Fully Homomorphic Encryption (FHE)* provides a new perspective for cloud computing security: it directly processes encrypted data so that attackers cannot get any sensitive data, even with a successful invasion of the system. As shown in Figure 1, the user encrypts and uploads the data to the cloud server, while the server fulfills the computation directly on the encrypted data. Then, the user decrypts the downloaded data to get the results. Compared with other privacy-preserving computing technologies (e.g., Federal Learning [40]), FHE is general enough to implement different kinds of applications, such as machine learning [30], [42], [54], information retrieval [63] and genome analysis [36].

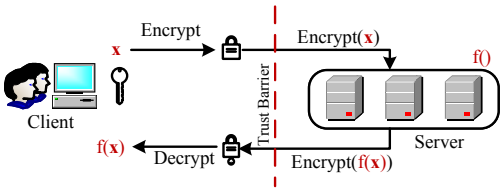

Fig. 1. The conceptual workflow of FHE.

However, the high performance overhead blocks the widespread adoption of FHE in real systems. For example, even with highly optimized FHE libraries and high-end CPUs, FHE computations still cost $10^4\times$ to $10^5\times$ more time than equivalent applications based on undecrypted data.

To bridge this performance gap, a series of optimization solutions have been proposed based on different kinds of hardware. 100x [33] is the first high-performance CKKS implementation on GPGPU, which supports `Bootstrap` operations. Unfortunately, it suffers from a lack of hardware support for the modulo calculations. HEAX [56] is proposed to accelerate CKKS on FPGA, which effectively improves the performance of NTT and modulo operations. However, due to the limited on-chip resources, HEAX can only support a small workload that does not require `Bootstrap` operations. To meet the requirements of a large scale workload, a series of ASIC accelerators have been proposed, including the F1+ [57], the CraterLake [58], the BTS [38] and the ARK [35]. These accelerators significantly improve the performance of all FHE operations, which makes it possible to utilize FHE in real workloads. However, all of these works rely on the over-expensive large-scale on-chip buffer or register file (RF), which impedes the popularity of these accelerators. For ex-

Corresponding Author: Mingzhe Zhang (zhangmingzhe@iie.ac.cn). This work is supported in part by National Natural Science Foundation of China (Grant No.62002339 and No.62125208), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB44030200), and Joint Funds for Smart Computing of Natural Science Foundation of Shandong Province (Grant No.ZR2019LZH014).

ample, F1+ [57] and CraterLake [58] require 256MB on-chip RFs, while BTS [38] and ARK [35] use 512MB RFs.

Since high-end GPGPUs have been widely equipped in the cloud servers, it will be very attractive if FHE acceleration can be effectively performed on GPGPUs: on the one hand, using GPGPU to accelerate FHE operations requires very limited economic cost; on the other hand, since most cloud computing tasks (e.g., machine learning, data analysis, etc.) are fulfilled on GPGPUs, a GPGPU-based FHE acceleration solution helps to simplify its integration with other applications. In this paper, we first analyze the fundamental reason that limits FHE performance on GPGPU. Then, we propose three techniques to improve the performance of all kinds of FHE operations on GPGPU: 1) algorithm optimization for NTT (key kernel of FHE) to improve its hardware efficiency on the GPGPU; 2) NTT optimization for fulfilling the NTT kernels on the emerging TCU hardware; 3) data layout optimization for improving the throughput of the batching FHE operations.

We evaluate our proposed TensorFHE with real workloads and provide a comparison with previous works on CPU, GPGPU, FPGA and ASIC accelerators. The evaluation results show that TensorFHE provides significant performance improvement for the key kernel and the operations, which is a speedup of up to 397.1× and 1035.8× for *HMULT* and *HADD*, respectively. When considering overall performance for real workloads, TensorFHE achieves higher performance than the F1+ [57] on `LR` application, although slightly lower performance than other accelerators. Considering the implementation cost, we believe that the TensorFHE is a more competitive solution for accelerating FHE in the cloud server.

In summary, our contributions are as follows:

- We analyze the fundamental reasons from the micro-architectural level for the low performance of NTT, and then propose an optimization at the software level.
- We propose a novel algorithm optimization that allows high-accuracy-performing NTT on the TCU with limited accuracy hardware support.
- We optimize the data layout to fully utilize the potential data parallelism for batching FHE operations.
- We provide a detailed evaluation of TensorFHE from different perspectives. The evaluation results show that our TensorFHE provides significantly higher performance than previous works on CPU, GPU and FPGA. Moreover, TensorFHE achieves comparable or even higher performance when compared to ASIC accelerators.

II. Background

In this section, we first introduce the basics of *Number Theoretic Transform (NTT)*, which is the key module of FHE. Then, taking CKKS as an example, we briefly introduce the main concepts of FHE. Last, we briefly introduce the micro-architecture of TCU. For ease of understanding, we summarize all of the symbols used in this paper in Table I.

TABLE I
SYMBOLS AND NOTIONS USED IN THIS PAPER.

Symbol	Definition
Q	(Prime) moduli product $\prod_{i=0}^{L} q_i$
P	Special (prime) moduli product $\prod_{k=0}^{K} p_k$
L	Maximum (multiplicative) level
dnum	Decomposition number [31]
K	Number of special prime moduli
N	Degree of a polynomial
q_l	(Prime) moduli, where $0 \leq l < L$
p_k	Special (prime) moduli, where $0 \leq k < K$
ψ, ψ^{-1}	Root of unity of twiddle factor for (I)NTT

A. Number Theoretic Transform

Number Theoretic Transform (NTT) is a specialized form of *Discrete Fourier Transform (DFT)* and widely adopted in various cryptography applications. Unlike DFT, NTT uses the ψ as the primitive N-th root of unity to convert polynomials in a finite field of integers, where $\psi_N^N \equiv 1 \bmod q$ for a given N and a prime q. According to Fermat's Little Theorem [62], for the prime q, exists at least one primitive root g such that $g^{(q-1)} \equiv 1 \bmod q$, generating the primitive N-th root of unity $\psi_N = g^{(q-1)/N} \bmod q$. Therefore, the overall NTT algorithm can be formulated as

$$A_k = \sum_{n=0}^{N-1} (a_n \psi_{(N,q)}^{nk} \mod q), k \in [0, N) \tag{1}$$

where a_n indicates the n-th coefficient of the input polynomial, A_k indicates the k-th coefficient of the output polynomial, and $\psi_{(N,q)}$ indicates the primitive N-th root of the NTT's unity for Z_q. Similarly, the *inverse NTT (INTT)*, which is the inverse process of NTT, can be formulated as

$$a_k = \frac{1}{N} \sum_{n=0}^{N-1} (A_n \psi_{(N,q)}^{-nk} \mod q), k \in [0, N) \tag{2}$$

where $\psi_{(N,q)}^{-1}$ refers to the primitive N-th root of the INTT's unity for Z_q, and $\psi_N^{-1} = \psi_N^{(q-2)}$ in the light of Fermat's Little Theorem [62].

Based on NTT, the polynomial multiplication of $A(X)$ and $B(X)$ can be performed as $A(X) \cdot B(X) = INTT(NTT(A(X)) \odot NTT(B(X)))$. To avoid applying the NTT on a $2N$-length input with N zero-padding, a negative-cyclic convolution is used to maintain the multiplication in a polynomial ring $Z[X]/(X^N + 1)$ [55]. The coefficients of the result in the polynomial ring are the same as the output of the negative-cyclic convolution. Therefore, the polynomial multiplication between $A(X)$ and $B(X)$ can be formulated as

$$\begin{aligned} c &= \psi^{-1} \odot INTT(NTT(\bar{a}) \odot NTT(\bar{b})), \\ \Psi^{-1} &= \{1, \psi_{(2N,q)}^{-1}, \psi_{(2N,q)}^{-2}, ...\psi_{(2N,q)}^{-(N-1)}\} \end{aligned} \tag{3}$$

where the operator $\odot$ indicates the element-wise multiplication of the coefficient of two polynomials, the $\bar{a}$, $\bar{b}$ and $\bar{c}$ are the vectors composed by the coefficients of $A(\psi_{2N,q} \cdot X)$, $B(\psi_{2N,q} \cdot X)$ and $C(X)$. By merging the $\psi_{(2N,q)}$ and the Ψ^{-1} with NTT/INTT, the new formula integrated with negative-cyclic convolution can be represented as

$$A_k = \sum_{n=0}^{N-1} ((a_n \psi_{2N}^{2nk+n}) \bmod q). \tag{4}$$

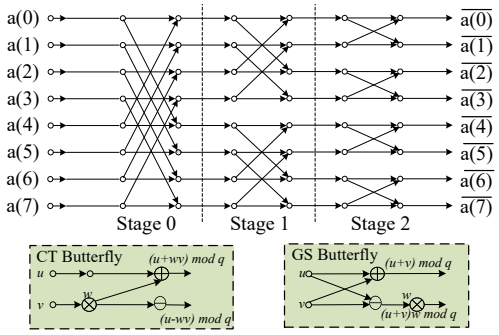

Fig. 2. An example of 8-point polynomial NTT/INTT with butterfly operation. Especially, the *CT Butterfly* is used in NTT, and the *GS Butterfly* is for INTT.

As shown in Eq. 4, the integer modular multiplication operation, which is the basic computation operation of NTT/INTT, produces high computational overhead [37]. To tackle this issue, the Cooley-Tukey [19] and Stockham [18] algorithms are leveraged to reduce the computation complexity from $O(N^2)$ to $O(Nlog_N)$, which reduces the amount of integer modular multiplication operation in NTT.

As shown in Figure 2, the Cooley-Tukey and Stockham algorithms use *butterfly operation* to calculate the two elements of interval $N/(2i)$ at $stage_i$, and the later $stage_{(i+1)}$ depends on the result produced by the prior $stage_i$, which causes the data-dependency issue between the neighboring stages. Prior works [13], [22], [24], [37] exploit the *split-radix* technique to relieve the dependency issues, which divide the N-point of NTT into k interleaved N/k-point NTTs (also known as the *radix-k NTT*). However, the radix-k NTT can hardly remove all dependency issues and inevitably requires extra synchronizations. In this work, we eliminate dependency issues by fulfilling the NTT algorithm with matrix multiplications.

B. CKKS Scheme

CKKS is an emerging FHE scheme with support for fixed-point real number [16], which is considered to be one of the most prominent FHE schemes for real-world tasks [30], [41], [42], [54]. In this paper, we focus on CKKS, but the proposed technique can also be applied to other FHE schemes, such as BFV [26] and BGV [7]. We will discuss the generality of our work in Section VII.

In CKKS, each $N/2$ input real numbers are encoded as

$$m(X) = \sum_{i=0}^{N-1} c_i X^i,\ N = 2^n,\ n \in [10, 18] \tag{5}$$

where $m(X)$ is a polynomial in plaintext with the cyclotomic polynomial ring $\mathbb{R}_Q = \mathbb{Z}_Q[X]/(X^N - 1)$. Each $m(X)$ contains N coefficients ($\{c_i\}$), which are integers moulded by Q. Q is a prime number with hundreds or even thousands of bits that relate directly to the HE ciphertext spaces. Then CKKS encrypts the $m(x) \in \mathbb{R}_Q$ into a ciphertext polynomial pair $(a(X), b(X))$ as

$$\begin{aligned} ct &= (a(X), b(X)) \\ &= (b(X) \cdot s(X) + m(X) + e(X), b(X)) \end{aligned} \tag{6}$$

where ct refers to the ciphertext, $s(X) \in R_Q$ refers to the secret key, $a(X) \in R_Q$ refers to the random polynomial and $e(X)$ refers to the small *Guassian Error polynomial* that guarantees the LWE security [16]. Based on the ciphertext, the FHE operations in CKKS can be summarized as follows:

- `HMULT(A,B)` fulfills the multiplication of two ciphertexts ($A(a_0(X), b_0(X))$ and $B(a_1(X), b_1(X))$) as ($a_0 \cdot a_1$, $a_0 \cdot b_1 + a_1 \cdot b_0$)+`KeySwitch`($b_0 \cdot b_1$).
- `CMULT(A,B)` fulfills the multiplication between ciphertext A($a_0(X), b_0(X)$) and plaintext B($a_1(X)$) in the manner of element-wise multiplication, which can be formalized as ($a_0(X) \cdot a_1(X)$, $b_0(X) \cdot a_1(X)$).
- `HADD(A,B)` adds the ciphertext A($a_0(X), b_0(X)$) to the other ciphertext B($a_1(X), b_1(X)$) in the manner of element-wise addition, which can be formalized as A($a_0(X)$, $b_0(X)$)+B($a_1(X), b_1(X)$)=($a_0(X) + a_1(X)$, $b_0(X) + b_1(X)$).
- `HROTATE(A,r)` circularly shifts the r-th ciphertext with the granularity of element-wise, which usually serves the accumulative operation of ciphertexts. `HROTATE(A, r)` can be formalized as A(0, $rotate(b_0, r)$)+ `KeySwitch`($rotate(a_0, r)$).
- `RESCALE(A)` updates the security level budget after the execution of `HMULT` and `CMULT`, which can be formalized as A($rescale(a_0), rescale(b_0)$).

The most significant performance bottleneck of CKKS is the overly high computational overhead. To tackle this issue, the following techniques are applied:

- **Residue Number System (RNS).** To enable modulo computation of the wide Q and coefficients, Chinese Remainder Theorem (CRT) is usually used to decompose the coefficients, which introduces huge computational overhead [25]. Therefore, the Residue Number System (RNS) [5] is represented to convert the ciphertext polynomial with the wide coefficients to L residue polynomials with 32-bit coefficients, each polynomial coefficient computed as $c_i \mod q_l$, where $0 \le i < N$ and $0 \le l < L$. In this way, the Full-RNS CKKS scheme significantly reduces the complexity of FHE operations [6].
- **Generalized Key-Switching (GKS).** The state-of-the-art GKS technique [31] reduces computational cost by balancing L, which decomposes the Q into $dnum$ slices ($\{Q_j\}_{0 \le j < dnum}$=$\{\prod_{i=j\alpha}^{(j+1)\alpha-1} q_i\}_{0 \le j < dnum}$, where $\alpha = (L+1)/dnum$). GKS allows that P=$\prod_{k=0}^{K-1} p_k$ only needs to be larger than $MAX_{0 \le j < dnum}(Q_j)$. Thus, the ciphertext computed in `keySwitch` can be decreased by adjusting $dnum$.

In this paper, we adopt Full-RNS together with generalized key-switching in our scheme to pursue the best performance.

C. Tensor Core Unit (TCU)

The TCU is a specialized component for accelerating *multiply and accumulate (MAC)* operations. It was first introduced to NVIDIA GPGPUs in the Volta architecture [50] and then improved in the Turing architecture [32] and the Ampere architecture [51]. As shown in Figure 3, each TCU is composed of two octets, while one warp simultaneously uses two TCUs. In one octet, each thread group owns a specific buffer for the LHS matrix (A Buf) and accumulation results (ACC Buf).

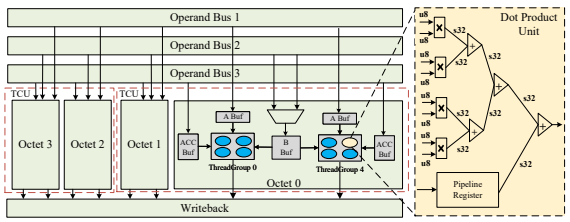

Fig. 3. Illustration of the TCU micro-architecture.

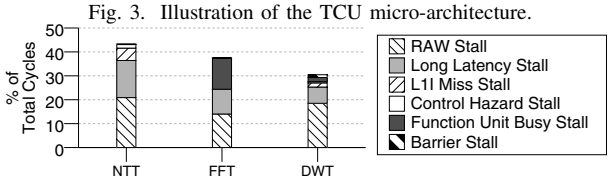

Fig. 4. GPGPU pipeline-stall breakdown of different algorithms based on *Butterfly* operations. The thread block sizes of NTT, FFT and DWT are 128, 192 and 256, respectively.

The two thread groups in the same octet share one RHS matrix buffer (B Buf) and select its source with a multiplexer.

Although a TCU achieves significantly improved performance [50], it can only support the low-precision computation (i.e., `FP16`, `BF16`, `INT8`, `INT4` and `INT1`). Taking the `UINT8` MAC as an example, each thread can access a four-by-four *Dot Product Unit (DPU)*, and each multiplexer in the DPU generates and sends the production to a `SINT32` accumulator for further computation. Due to the limited precision, TCUs cannot be directly utilized for FHE acceleration. This is the first work to accelerate FHE operations on TCUs.

III. Motivation

In this section, we provide a comprehensive understanding of running FHE on GPGPU at the micro-architecture level. We first quantitatively analyze the underutilization of hardware. Then, we present the challenges of using the emerging TCUs in FHE acceleration. Finally, we summarize the opportunities for a pure software solution to accelerate the FHE on GPGPUs.

A. Inefficient NTT Computation

To investigate the pipeline stall issue of NTT, we simulate an NVIDIA 1080Ti GPU on GPGPUSim [34] and run a state-of-the-art NTT implementation [22] on it. We also run FFT and DWT implementations from the Rodinia Benchmark Suite [11].Note that, since the GPGPU switches to the other warp when a pipeline stall occurs, here we consider only the stall cycles that cannot be hidden. The results are shown in Figure 4, and we can make the following observations:

- All kernels suffer from the pipeline stall. Especially, the proportion of the pipeline stall time is up to 43.2% for NTT.
- Not surprisingly, the most significant reason for the pipeline stall is the *Read-After-Write (RAW)* issue in all kernels, which can be inferred as the result of data-dependency between the neighboring stages in the *butterfly operations*. For the NTT kernel, the proportion of the RAW stall is 20.9%, which is 48.6% of its overall pipeline stalls.

Therefore, the key to boosting NTT on GPGPU is to reduce the RAW stall as much as possible.

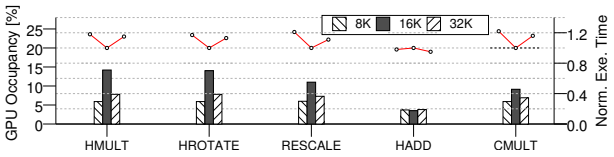

Fig. 5. Impact of threading on the GPGPU occupancy and performance for CKKS operations. The bars indicate the GPU occupancy (left), and the line charts indicate the Normalized Execution Time (right).

B. Low Computation Resource Utilization

As GPGPU utilizes a large amount of SIMD cores with the static pipeline, once a pipeline stall occurs, the scheduler directly switches to other pending threads to fully utilize the hardware resources. Therefore, increasing the thread number for each Streaming Multiprocessor (SM) would help to improve the hardware occupancy by better hiding the possible stalls. However, as the thread in one SM increases, the performance will be hurt due to there being more intense competition for the resources inside the SM.

To investigate this tradeoff, we run the instance of TensorFHE-NT as described in Table IV with no batching on the NVIDIA A100 GPGPU (see Table III for detail) and monitor the hardware occupancy and the operation performance. We increase the total number of threads from 8196 (8K) to 32768 (32K) to show its impact. Note that, we evaluate different thread number per SM and report the best performed results, which contains 512 threads in each SM. As shown in Figure 5, we can make the following observations:

- Not surprisingly, as the thread number increases from 8K to 16K, the occupancy of all CKKS operations grows. In particular, for the most frequently used `HMULT`, the GPGPU occupancy is up to 14.3%. Accordingly, the execution time of all operations is also reduced.
- However, as the thread number continues increasing to 32K, the GPGPU occupancy of all operations degrades. This is because each thread gets fewer data and the total number of memory accesses increases, which leads to less efficient bandwidth utilization. Also, the execution time is increased. The only exception occurs in `HADD` since the workload for each thread is minor and the execution time ($< 0.1\mu s$) is significantly affected by other factors.
- Moreover, the highest GPGPU occupancy for all kinds of operations is lower than 15%. When considering the performance, the under utilization of GPGPU is unacceptable: with the best performance of all operations, GPGPU occupancy is around 10% and even lower than 5% for the `HADD`.

Ideally, we want a system that fully utilizes the hardware resources to achieve performance that is as high as possible. But state-of-the-art CKKS fails on this goal.

C. Ineffective Usage of Emerging Hardware

As we introduced in Section II-C, modern GPGPU uses TCUs to accelerate the MACs and achieves great success with AI workloads [27]. However, the limited precision support makes it hard to accelerate the arithmetic kernels of CKKS (i.e., NTT/INTT) with TCUs. For example, in the state-of-the-art implementation of CKKS, the NTT/INTT kernel is based

on the computation for INT32, while the TCUs only support up to INT8. Besides, GPGPU lacks efficient hardware support for the modulo operation, which also significantly affects the performance of CKKS on GPGPU [33].

D. Our Opportunities

Based on the above observations, we conclude our opportunities for boosting FHE on GPGPU as follows:

- As discussed in Section III-A, the NTT computation suffers from serious RAW stalls and low efficiency of modulo operations. Therefore, we could optimize the NTT algorithm to remove the RAW stalls and the excessive modulo operations.
- As discussed in Section III-C, modern GPGPU contains emerging TCUs for high-performance MAC computations. To fully utilize the powerful TCUs, algorithm optimization for NTT is necessary, which aims to apply high-accuracy NTT based on the hardware with limited accuracy support.
- As discussed in Section III-B, current thread-level parallelism cannot sufficiently utilize the abundant hardware resources of GPGPU. Therefore, we exploit the overall performance improvement of FHE with batching techniques. RebuttalChangeHowever, Ref [33] observes that for all CKKS operations, the bandwidth is more intensive than the computation resources, which limits the capability of batching execution. To tackle this issue, we break the CKKS operations into a series of reusable kernels, which balances the bandwidth and computation requirement of each kernel. Such a hierarchical reconstruction model also helps to better understand and optimize the complex CKKS scheme.

Consequently, we propose a series of optimizations and finally integrate them together. This, in turn, improves the performance of FHE execution on GPGPU. In the next section, we introduce our detailed scheme: *TensorFHE*.

IV. TensorFHE

Based on the analysis made in Section III, we propose *TensorFHE*, which accelerates FHE operations on GPGPU. TensorFHE is based on a hierarchical reconstruction of CKKS, which decomposes the CKKS operations into a series of reusable arithmetic kernels. For the most time-consuming NTT/INTT kernels, TensorFHE first uses an optimized algorithm to reduce pipeline stalls. It then further improves the algorithm to exploit accelerating NTT/INTT kernels by using the emerging TCU. For the rest of the kernels, TensorFHE conducts the paralleled algorithms on CUDA cores. Moreover, to fully utilize the potential computational and data-level parallelism of GPGPU, TensorFHE also introduces an optimization scheme for batching multiple FHE operations. As a result, TensorFHE provides significant performance improvement for CKKS on GPGPU with pure software optimizations. In the rest of this section, we introduce TensorFHE in detail.

A. Hierarchical Reconstruction of CKKS

To better understand and optimize the CKKS scheme, TensorFHE uses a hierarchical model to reconstruct the CKKS scheme. As shown in Table II, each FHE operation can be decomposed into multiple reusable arithmetic kernels. Overall, there are seven involved kernels as follows:

TABLE II
Hierarchical reconstruction model of CKKS.

Operation	Description	Composing Kernels
HMULT	Multiply two ciphertexts.	NTT, Hada-Mult, Conv, Ele-Add
CMULT	Multiply ciphertext with plaintext.	Hada-Mult, Ele-Add
HROTATE	Roatate ciphertext.	NTT, Hada-Mult, Ele-Add, Conv, ForbeniusMap
RECALE	Reduce the security level of ciphertext.	NTT, Ele-Sub
HADD	Add two ciphertexts.	Ele-Add

Algorithm 1: KeySwitch $([\mathrm{d}]_{C_l}, \mathbf{evk})$

Require: $l \leq (L-1)$
$\overrightarrow{d} \leftarrow$ Dcomp$([\mathrm{d}]_{C_l})$
$[d_j]_{D_\beta} \leftarrow$ModUp$(\overrightarrow{d})$
$([c_0]_{D_\beta}, [c_1]_{D_\beta}) \leftarrow$ Inner-product$([d_j]_{D_\beta}, \mathbf{evk})$
$([c_0]_{D_l}, [c_1]_{D_l}) \leftarrow$ (ModDown$([c_0]_{D_\beta})$, ModDown$([c_1]_{D_\beta})$)
return $([c_0]_{D_l}, [c_1]_{D_l})$

- ***NTT*** transforms coefficient represented polynomial into point-value representation to accelerate the polynomial multiplication. It can be formulated as $A_k = \sum_{n=0}^{N-1}(x_n \psi_{(N,p)}^{nk} \mod q)$, where $0 \leq k < N$ [15].
- ***Hadamard Multiplication (Hada-Mult)*** can be formulated as $c = (a \circ b) \mod q$ [15]. The $a \circ b$ indicates the element-wise product of two polynomials represented as vectors (also known as *Hadamard product*) [17].
- ***Element-wise Addition(Ele-Add)*** and ***Element-wise Subtract (Ele-Sub)*** can be formulated as $c = (a \oplus b) \mod q$ and $c = (a \ominus b) \mod q$, respectively [15]. Here we use $a \oplus b$ and $a \ominus b$ to indicate the element-wise addition and subtraction for the two polynomials represented as vectors [15].
- ***ForbeniusMap*** performs the Frobenius map function for the polynomial by index r under the NTT domain [31]. For every $a^{(i)}=(a_j^{(i)})_{j\in[0,N-1]}$ in $\{a^{(i)}\}_{i\in[0,l]}$, it generates $a'^{(i)}=((a^{(i)}_{\pi_r^{-1}(j)})_{j\in[0,N-1]}$, where $\pi_r(x)=([5^r(2x+1)]_{2N}-1)/2$, which is a permutation operation. Then, this kernel will return $a'=(a'^{(i)})_{i\in[0,l]}$.
- ***Conjugate*** indicates the conjugation of the coefficients in the polynomial with the modulus q [47]. For the polynomial $\mathbf{A}$ represented by coefficients, the conjugate representation of polynomial is $\bar{A}$.
- ***Fast basis Conversion*** Kernel (Conv) converts a set of residue polynomials to another set whose prime moduli is different from the former. It is the key operation of ModUp, ModDwon and ModRaising [15].

Based on the above arithmetic kernels, all CKKS operations can be composed as follows:

- **keySwitch** refreshes the secret key of the ciphertext and maintains its precision, which is widely adopted in several FHE operations. As shown in Alg. 1, the keySwitch mainly includes the ModUp, the ModDown, the Dcomp and the Inner-product. For the ModUp and the ModDown, we

Algorithm 2: HMULT $(ctx_0, ctx_1,$ **evk**$)$

Require: $ctx_0 \leftarrow (a_0, b_0)$, $ctx_1 \leftarrow (a_1, b_1)$
$d_2 \leftarrow$ **Hada-Mult**(a_0, a_1), $d_0 \leftarrow$ **Hada-Mult**(b_0, b_1)
$d_1 \leftarrow$ **Ele-Add**(**Hada-Mult**(a_1, b_0), **Hada-Mult**(b_1, a_0))
$(c_0^{'}, c_1^{'}) \leftarrow$ KeySwitch$(d_2,$**evk**$)$
return $\text{ctx}_{mult} = ($**Ele-Add**$(d_1, c_0^{'})$, **Ele-Add**$(d_0, c_1^{'}))$

Algorithm 3: CMULT (ctx, ptx)

Require: $ctx \leftarrow (a_0, b_0)$, ptx$\leftarrow (pa)$
$d_0 \leftarrow$ **Hada-Mult**(a_0, pa)
$d_1 \leftarrow$ **Hada-Mult**(b_0, pa)
return $\text{ctx}_{mult} = (d_0, d_1)$

Algorithm 4: HROTATE $(ctx, r,$**evk**$_{\mathbf{rot}})$

Require: **ctx** $\leftarrow (a, b)$
$a^{'} =$ FrobeniusMap(a, r)
$b^{'} =$ FrobeniusMap(b, r)
$(a^{''}, b^{''}) \leftarrow$ KeySwitch$(a^{'},$**evk**$_{\mathbf{rot}})$
return $\text{ctx}_{rot} = (a^{''},$ **Ele-Add**$(b^{'}, b^{''}))$

Algorithm 5: HADD (ctx_0, ctx_1)

Require: $ctx_0 \leftarrow (a_0, b_0)$, $ctx_1 \leftarrow (a_1, b_1)$
$d_0 \leftarrow$ **Ele-Add**(a_0, a_1)
$d_1 \leftarrow$ **Ele-Add**(b_0, b_1)
return (d_0, d_1)

directly use the method introduced in Ref [31], which can be composed by multiple *NTT*s and *Conv*s. For the Dcomp, which represents the RNS decomposition operation, it can be implemented with minor modified *Conv* [33]. Inner-product is the key operation in KeySwitch; it is implemented with multiple *Hada-Mult*s and *Ele-Add*s.

- **HMULT** and **CMULT** can be implemented with *Hada-Mult* and *Ele-Add* (as shown in Alg. 2 and 3). Note that the HMULT requires KeySwitch to reduce the ciphertext precision loss.
- **HROTATE** is composed by *ForbeniusMap* and *Ele-Add*, which also needs Keyswitch (as shown in Alg. 4).
- **HADD** requires only **Ele-Add** kernels (as shown in Alg. 5).
- **RESCALE** consists of multiple *NTT* and *Ele-Sub* (Alg. 6).
- **Bootstrap** includes four stages: the SlotToCoeff, the ModRaising, the CoeffToSlot and the Sine Evaluation (as shown in Figure 6). In this work, the Bootstrap is implemented according to the *slim botstrapping* [12], which reorders the stages and reduces the computational overhead of the SlotTocoeff. In the SlotToCoeff and CoeffToSlot stages, we use the *Baby-Step Giant-Step (BSGS)* [59] algorithm to fulfill the DFT computation, which is the most time-consuming operation and composed by multiple CMULT, HMULT and HROTATE operations. To improve BSGS performance, we adopt the *Faster Homomotrphic DFT algorithm* [14], which significantly reduces the requirement for the HROTATE and the CMULT. Besides, we use the *Taylor Polynomial Approximation* [8] to approximate the *sin(x)* function [31].

Based on this reconstruction, TensorFHE accelerates the CKKS operations with large-scale parallelism on GPGPU for all kernels. Especially for the NTT/INTT, which is the most time-consuming kernel [3], [45], TensorFHE exploits to accelerate it with the CUDA cores and the TCUs, which will be introduced in the rest of this section.

B. NTT Optimization for CUDA Core

According to Section II-A, the NTT algorithm can be formulated as Eq. 4 and implemented in the form of the *butterfly algorithm* [43]. However, the performance of the *butterfly algorithm* suffers from the frequent pipeline stalls caused by the RAW dependency issue. Besides, the large amount of modulo operations in Eq.4 also consumes a large amount of time due to the GPGPU's lack of efficient hardware support for modular arithmetic.

To overcome these issues, we use a transform technique to optimize NTT. Considering the modular arithmetic's compatibility with addition, Eq. 4 can be transformed as

$$A_k = \left(\sum_{n=0}^{N-1} a_n \psi_{(2N,q)}^{2nk+n}\right) \mod q, a_n \in a. \quad (7)$$

Such transformation converts the NTT from the butterfly algorithm to matrix-vector multiplication as

$$A = (W_{N\times N} \times a^T) \mod q, w_{ij} = \psi_{(2N,q)}^{2ij+j} \in W_{N\times N}. \quad (8)$$

Such transformation can benefit the NTT performance in the following aspects:

- **Data Reuse.** According to the CKKS definition, the *twiddle factor matrix* is determined by the FHE parameters, such as N and q. Therefore, for one CKKS instance, the *twiddle factor matrix* $W_{N\times N}$ can be pre-computed in the initialization and reused by all NTT operations. This can significantly reduce the computational and storage overhead.
- **Hardware Efficiency.** The matrix-vector multiplication has good memory locality and can be easily paralleled on GPGPU with high pipeline efficiency.
- **Modulo Reduction.** Only one modulo operation is required for each A_k, and a large amount of time can be saved due to the modulo reduction. Note that, since we are reducing the number of modulo operations, more memory space is required to store the longer temporary values. In this paper, we use a 64-bit integer as the accumulative variable, which does not meet the overflow issue until $N \leq 2^{18}$.

However, since the length of the polynomial (N) usually ranges from 2^{10} to 2^{18}, the scale of the *twiddle factor matrix* ($W_{N\times N}$) and the input vector (a) will be extremely large, which leads to over-high computation and memory overhead. To tackle this issue, we adopt the Cooley-Tukey Recursive algorithm [29], which transforms the input vector (a) to an input matrix ($a_{N_1\times N_2}$, where $N = N_1 \times N_2$). Accordingly, the scale of the *twiddle factor matrix* is also reduced to $N_1 \times N_1$. In this way, the NTT can be further transformed as

$$\mathbf{A} = ((\mathbf{a_{N_1\times N_2}} \times \mathbf{W_1})^T \odot \mathbf{W_2}) \times \mathbf{W_3} \mod q \quad (9)$$

where the scale of W_1, W_2 and W_3 is $N_1 \times N_1$, $N_1 \times N_2$ and $N_2 \times N_2$, respectively. Especially, the elements of the three matrices can be represented as $\psi_{(2N_1,q)}^{2ij+j}$, $\psi_{(2N,q)}^{2ij+j}$ and $\psi_{(2N_2,q)}^{2ij}$.

In this way, the complex NTT is transformed to three sequential matrix-matrix multiplications. Although the time complexity increases, the transformed algorithm could still benefit from better utilization of parallelism.

Algorithm 6: RESCALE (**ctx**)

Require: $ct \leftarrow ([a]_{C_l}, [b]_{C_l})$

$a^{'(j)} \leftarrow [q_l^{-1}(\textbf{Ele-Sub}(a^j, \textbf{NTT}([\textbf{INTT}(a^{(l)})]_{q_j})))]_{q_j}, j \in [0, l-1]$

$b^{'(j)} \leftarrow [q_l^{-1}(\textbf{Ele-Sub}(b^j, \textbf{NTT}([\textbf{INTT}(b^{(l)})]_{q_j})))]_{q_j}, j \in [0, l-1]$

return $([a^{'}_{C_{l-1}}], [b^{'}_{C_{l-1}}])$

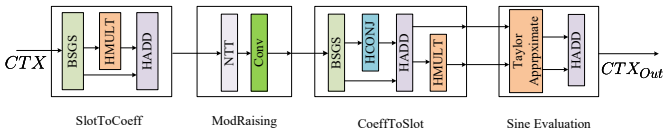

Fig. 6. `Bootstrap` workflow composed by various kernels.

C. NTT Optimization for Tensor Core

We further exploit to accelerate the NTT kernel with the emerging TCUs, which is considered to be the most powerful unit for MACs in the GPGPU. As mentioned in the above subsection, the NTT kernel is transformed into three sequential matrix multiplications, which provides the possibility of using TCUs. However, all matrix elements used in the above mentioned NTT kernel are UINT32, while TCU supports only the low-precision arithmetic operations (i.e., up to 8-bits for integer and 16-bits for floating-point data). We use a *segment-fusion* scheme to fulfill the GEMM without precision loss to overcome this issue.

Figure 7 presents the segmentation process. For each 32-bit integer element $m_{i,j}$ in the matrix M, we segment every consecutive 8-bits into an 8-bit integer. Then, each 8-bit integer is distributed to the corresponding place of $m_{i,j}$ in the segmented matrices. The matrix selection is according to the original location of the 8-bit integer. In this way, we can get four matrices composed by 8-bit elements ($\overline{M_0}$, $\overline{M_1}$, $\overline{M_2}$, $\overline{M_3}$), which can be processed by the TCUs.

Since the *twiddle factor matrices* can be reused in all NTT kernels, their segmentation can be fulfilled as the pre-processing of the TCUs-based NTT kernels. The workflow of the TCUs-based NTT kernels is as shown in Figure 8; it is composed of the following sequential stages:

1) Stage1-Input Matrix Segmentation: In this stage, we segment the input $a_{N_1 \times N_2}$ (mentioned in Eq. 9) into four small-scale matrices (denoted as T_0 to T_3) according to the process as shown in Figure 7. This stage is fulfilled with parallel threads on the CUDA cores; each thread fetches four consecutive 32-bit elements with the aim of maximizing global memory bandwidth utilization. Note that, to improve the efficiency of the subsequential GEMMs, all output matrices (T_0 to T_3) are stored in a column-major.

2) Stage2-TCU GEMM for $a_{N_1 \times N_2} \times W_1$: Based on T_0 to T_3, we fulfill the $a_{N_1 \times N_2} \times W_1$ on the TCUs. In this stage, the workload distributed to the TCUs can be formulated as

$$O_{ij} = W_{1i} \times T_j, \quad i, j \in [0, 3] \tag{10}$$

where W_{1i} indicates the segmentation matrices of W_1. Overall, there are 16 matrix multiplications to be fulfilled. To maximize computation efficiency, we leverage the workload concurrency by assigning each GEMM to a separate stream [28]. Besides, we also use the open-source CUTLASS library [49] to implement all matrix multiplications. Note that, since the datatype of

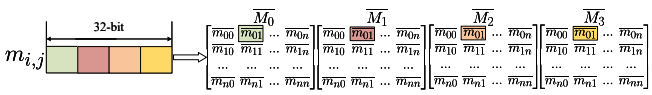

Fig. 7. Illustration of the segmentation process for the matrix based on a 32-bit integer.

TCU output is restricted as *s32* and the higher 16-bits are filled with zeros, it is risk-free for overflow in this stage. Besides, the output matrices O_{ij} are stored in row-major.

3) Stage3-Fusion and GEMM with W_2: The output matrices of `Stage2` (O_{ij}) are firstly fused as one by using the *Booth multiplication algorithm* [44], which treats the corresponding elements from every matrix as partial accumulations. The elements of the fused matrix are in the type of u32. After the fusion, we conduct a *Hadamard multiplication* for the fused matrix and the W_2. Then, we segment the result matrix of the *Hadamard multiplication* into four *u8* matrices stored in column-major (T'_0 to T'_3) for the next stage.

4) Stage4-TCU GEMM with W_3: Similar to `Stage2`, we fulfill the $O'_{ij} = T'_j \times W_{3i}, i, j \in [0, 3]$ on the TCUs. Each matrix multiplication is assigned to a separate stream to maximize the hardware efficiency.

5) Stage5-Fusion and Output: Similar to `Stage 3`, the output matrices of `Stage4` ($O'_i j$) are fused into one *u32* matrix, which is stored in row-major, using the *Booth multiplication algorithm*. Then, the fused matrix is moduloed element-wise by q with output as the NTT result. Note that, for the `INTT` kernel, an extra modular multiplicative inverse of N under q is multiplied element-wise with the result matrix before output.

Overall, `Stage1`, `Stage3` and `Stage5` are fulfilled on the CUDA cores with parallel threads, while the `Stage2` and `Stage4` are fulfilled on TCUs. In this way, we can significantly improve the performance of the NTT kernel by fully utilizing the emerging hardware resources in GPGPU.

D. Operation-Level Batching

As we discussed in Section III-B, the GPGPU is seriously underutilized when running CKKS operations. Therefore, we exploit to improve the overall performance of a real workload and hardware utilization by batching multiple FHE operations.

With operation-level batching, all running kernels process the data with the same L, since they can reuse the same *twiddle factor matrix* for NTT. However, the improperly designed data layout may impair the effectiveness of the batching. As shown in Figure 9(a), m+1 CKKS operations are batched and their required data are originally stored in the manner of $(B,\ L,\ N)$. For each batching operation (i.e., B_0, ..., B_m), there are L data entries stored in a contiguous address space (denoted as a group) and the size of each entry is $N \times$ 32-bits. During the execution, the corresponding data entries with the same L value from all groups are packed. Because the data entries are from the discontinuous memory space, performance of the batching execution suffers from low bandwidth utilization during the data packing.

When fully utilizing the data parallelism, we optimize the data layout for the operation-level batching. As shown in

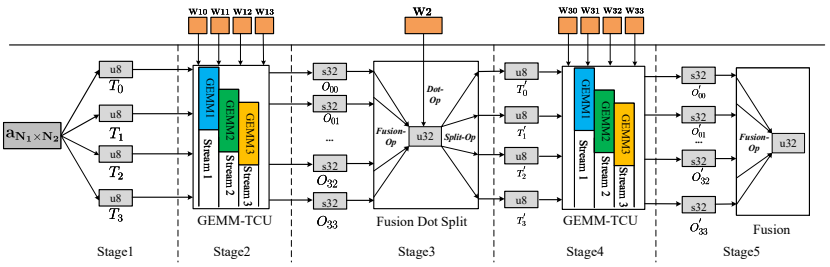

Fig. 8. Overall workflow of the NTT based on GEMM-TCU.

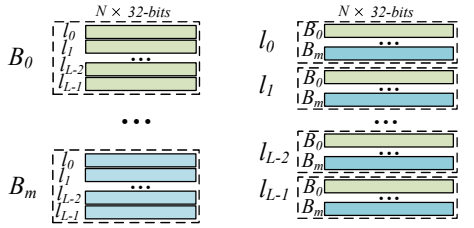

Fig. 9. Data layout optimization in TensorFHE.

TABLE III
PLATFORM CONFIGURATIONS.

CPU	Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz
GPGPU	NVIDIA A100-SXM-40GB
Memory	512GB
Misc.	CUDA 11.0; Pytorch 1.7.0; cuPy 9.6.0

Figure 9(b), we reorganize the data layout in the manner of $(L,\ B,\ N)$, which stores all data entries with the same L in continuous memory space. Thus, the batched operations can load the continuous data block (in size of $(m+1) \times N \times 32\text{-}bits$) for data packing, which helps to maximize bandwidth utilization. In this way, basic kernels can automatically generate a large amount of CTAs with the packed data and utilize as many GPGPU resources as possible.

E. TensorFHE Implementation

By integrating the kernels and the above mentioned techniques, we can get an implementation of TensorFHE, which includes the following layers:

- **API Layer** runs on the CPU, which collects and decomposes the requests for FHE operations from the user applications. The decomposed requests are in the manner of workflow, which consists of the basic kernels. Then, the API layer automatically generates the best batch size for the different involved kernels according to the hardware resources. Finally, the API layer sequentially invokes the kernels in the workflow with proper batch size.
- **Kernel Layer** consists of various arithmetic kernels that run on the GPGPU. The kernel layer receives the invoking request from the API layer and returns the results of the requested operations. Note that the intermediate results of the kernels are stored in the GPGPU VRAM.

Based on such implementation, TensorFHE fully utilizes the hardware resources and efficiently supports all CKKS operations (including `Bootstrap`) on a single GPGPU.

V. METHODOLOGY

We implement our proposed TensorFHE based on CUDA 11.0 [48] and PyTorch 1.7 [53]. Then, we evaluate the TensorFHE on a high-end server equipped with one NVDIA A100 GPGPU. The detailed configurations of the platform are as shown in Table III. Besides, we use NVIDIA Nsight [52] to monitor the SM occupancy during the execution.

To fully evaluate our proposed TensorFHE, we compare it with a series of previous works based on various devices. The involved designs are as shown in Table IV. To compare

TABLE IV
COMPARED SCHEMES.

Type	Design	Description
CPU	Baseline [58]	AMD Ryzen 3975WX
GPU	PrivFT [1]	NVIDIA Tesla V100, VRAM=16GB
GPU	100x [33]	NVIDIA Tesla V100, VRAM=16GB
FPGA	HEAX [56]	Stratix 10 GX 2800, 11.7KB RFs
ASIC	F1+ [57]	16 compute cluster (16 NTT FUs, 32 Mul FUs, 16 Add FUs), 64MB RFs.
ASIC	CraterLake [58]	1× CRB FUs, 2×NTT FUs,5× Mul FUs and 5× Add FUs, 256MB RFs
ASIC	BTS [38]	2048 × (NTT FUs,Add FUs Mul FUs), 512MB RFs
ASIC	ARK [35]	4× BConv FUs, 4×NTT FUs, 4× Auto FUs, 4× MAD FUs 512MB RFs
GPGPU (A100)	TensorFHE-NT	Batching, NTT with *butterfly operations*
	TensorFHE-CO	Batching, NTT with GEMMs
	TensorFHE	Batching, NTT with TCUs

TABLE V
CKKS PARAMETERS USED IN THE EXPERIMENTS.

	N	L	K	logPQ	batch_size
Default	2^{16}	44	1	1306	128
ResNet-20	2^{16}	29	1	840	64
Logistic Regression	2^{16}	38	1	1092	64
LSTM	2^{15}	25	1	728	32
Packed Bootstrapping	2^{16}	57	1	1624	32

TensorFHE with works based on CPU, GPGPU and FPGA, we measure the performance of the key kernel and the operations, including `NTT`, `HADD`, `HMULT`, `HROTATE` and `Bootstrap`. For the comparison with ASIC accelerators, we evaluate the TensorFHE by using four CKKS programs with state-of-the-art implementations and measure the overall execution time. Note that we directly collect data from the literature for the previous works, and use a dash '-' to represent data not mentioned in the literature.

The programs used for the comparison with ASIC accelerators are as follows:

- **ResNet-20 [42]**. We implemented this DNN model for image recognition with FHE. In our experiment, 64 encrypted images are packed as the input.
- **Logistic Regression [30]**. We test the HELR algorithm with 16384 samples, and batch-encode 128 samples in one polynomial. We execute it for 14 iterations, which three bootstrapping operations are required.
- **LSTM [54]**. This is an NLP model for analyzing the contextual information of a sentence. We implement this model in FHE as [54], which includes 128 cells, and the dimension for each word embedding used in this model is 128. We pack 32 sentences in parallel as the input.

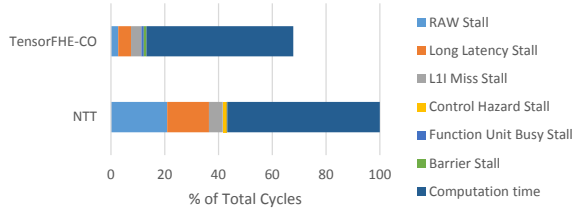

Fig. 10. Normalized pipeline execution time breakdown comparison of the NTT implementations.

• **Packed Booststrapping [46]**. This example performs packed booststrapping operations with the same configuration as [58]. The ciphertext with $N = 64k$ restores its security level to L=57. In our experiment, we perform 32 ciphertexts in parallel for the bootstrapping operations.

Table V presents the CKKS parameters used in the experiments. Without specific description, we use parameters in `Default` for all experiments.

VI. Results

In order to evaluate the effectiveness of TensorFHE, we provide a series of comparisons between TensorFHE and various previous works. For the comparison with the works on CPU, GPGPU and FPGA, we focus on the performance of FHE operations and the throughput of the critical kernels. On the other hand, we measure the performance of the real workload for comparison with the ASIC accelerators. We also provide a sensitivity study of the different FHE parameters.

A. The effectiveness of NTT optimization

To evaluate the effectiveness of NTT optimization in TensorFHE, we run the optimized NTT kernel (used in the TensorFHE-CO) on the simulation platform as introduced in Section III-A, and then compare the execution time breakdown to the state-of-the-art implementation [22]. As shown in Figure 10, TensorFHE significantly reduces the *RAW stall* by 18.1% and the *Long Latency Stall* by 10.8%. Note that, although transformation from *butterfly operations* to the matrix operations increases the computation time by 1.2%, the overall performance of NTT is still improved by 32.3% due to the reduction of the various pipeline stalls. Besides, we also validate the correctness of the optimized NTT kernel by using successive NTT and INTT kernel. We compare the NTT input with the INTT ouput, and find that the two values are exactly the same, which proves the correctness of our optimization.

B. TensorFHE vs. CPU, GPGPU and FPGA

1) Performance: As shown in Table VI, TensorFHE achieves significant speedups over the implementation based on CPU, which is 397.1× faster with `HMULT` and up to 1035.8× faster with `HADD`. Moreover, for the `HMULT` and the `HROTATE`, which involve expensive NTT computations, TensorFHE-NT achieves the speedup over 100x [33] (the state-of-the-art CKKS implementation on GPGPU) for 1.04× and 1.02×, respectively. Such improvement proves that hardware efficiency significantly impacts FHE performance, which is mainly caused by the batching execution. Besides, TensorFHE-CO helps to achieve 1.35× and 1.41× speedup over 100x [33] for the `HMULT` and the `HROTATE` operation, which is directly improved by reduction of the RAW stalls and the modulo operations. Besides, TensorFHE performs 2.49× and 1.84× speedup over TensorFHE-NT and TensorFHE-CO, which shows the impact of emerging TCUs on NTT operations. Moreover, to directly compare the performance, we re-run the TensorFHE on NVIDIA V100 (same as 100x [33]), and observe that the TensorFHE still performs faster than 100x [33], which is up to 5.26× speedup for the `RESCALE`. Furthermore, as shown in Table VII, TensorFHE achieves 1.3× speedup over 100x for `Bootstrap`, which is the key to fulfilling complex FHE workloads.

TABLE VI
Operation Delay Comparison of TensorFHE, CPU and GPGPU on micro FHE benchmarks (ms)

	HMULT	HROTATE	RESCALE	HADD	CMULT
CPU [33]	338s	330s	18611	3609	3356
PrivFT [1]	7153	-	208	24	21
100x [33]	2227	2154	81	26	22
TensorFHE-NT	**2124**	**2111**	**35**	**6**	**7.7**
TensorFHE-CO	**1651.2**	**1523.2**	**9.2**	**6**	**7.7**
TensorFHE(V100)	**1296.6**	**1254.4**	**15.4**	**10.2**	**11.5**
TensorFHE(A100)	**851**	**852**	**7.7**	**6**	**7.7**

TABLE VII
Execution Time Comparison for the Bootstrap (in seconds) ($N=2^{16}$, L=34, dnum=5 and batch_size = 128).

CPU [33]	GPGPU baseline [33]	100× [33]	Tensor FHE-NT	Tensor FHE-CO	Tensor FHE
10168	54904	42016	76731	70762	32058

We also compare TensorFHE to the state-of-the-art implementation of FHE on FPGA. As shown in Table VIII, TensorFHE achieves 4.9× speedup on average for the (i)NTT kernel. For the `HMULT` operation, TensorFHE can also achieve 1.46× speedup for Set_C. However, with the small polynomial length of Set_A, TensorFHE is slower than HEAX by about 10%, which implies that algorithm complexity reduction provides more advantages than massive parallelism for workloads requiring less computation.

2) Execution Time Breakdown: Table 11 presents the execution time breakdown of different TensorFHE operations. Not surprisingly, the NTT kernels occupy the most significant proportion in `HMULT` and `HROTATE`, that is, 92.1% and 95.4%, respectively. This indicates that, though current TensorFHE provides remarkable performance improvement for NTT/INTT kernels, there are still more opportunities for further improvement. On the other hand, the non-NTT kernels only take a small part of the time, which indicates that the fully parallel scheme using TensorFHE has already achieved good performance for the non-NTT kernels.

3) Hardware Utilization: As we discussed in Section III, FHE performance suffers from low GPGPU occupancy. As shown in Table IX, TensorFHE achieves remarkable improvement for hardware occupancy by using the batching techniques. Especially, for the `HMULT` and the `HROTATE`, the GPGPU occupancy is over 90%. Note that for the GPGPU occupancy calculation, we consider the TCUs for the `HMULT` and the `HROTATE`, while only considering the CUDA cores

TABLE VIII
PERFORMANCE COMPARISON FOR TENSORFHE AND HEAX. TAKING THE THROUGHPUT OF THE KEY KERNELS AND OPERATION AS METRIC.

		Set_A	Set_B	Set_C
#NTT/second	CPU [56]	7222	3437	1631
	HEAX [56]	195313	90144	41853
	TensorFHE	910134	449974	209337
#INTT/second	CPU [56]	7568	3539	1659
	HEAX [56]	195313	90144	41853
	TensorFHE	913267	449084	209178
#HMULT/second	CPU [56]	420	84	15
	HEAX [56]	97656	22536	2616
	TensorFHE	88048	27564	3825

Set_A: N=2^{12}, log_{pq} = 108, K=2; Set_B: N=2^{13}, log_{pq} = 217, K=4; Set_C: N=2^{14}, log_{pq} = 437, K=8.

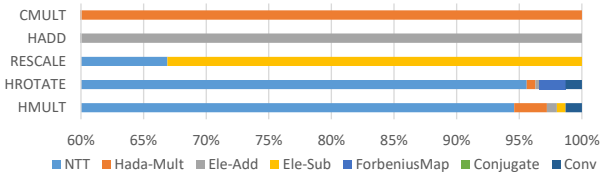

Fig. 11. Execution Time Breakdown for the FHE Operations.

for the other operations.

C. Evaluation with Real Workload

1) Performance: As shown in Table X, for some specific applications, TensorFHE provides comparable performance with the state-of-the-art FHE accelerators based on ASIC. Especially, for the LR, TensorFHE achieves 1625.6× and 2.9× speedup over the CPU and F1+, respectively. When compared with the other ASIC accelerators (i.e., BTS, ARK and CraterLake), there is still a performance gap, up to 40×. However, since TensorFHE is a high-performance FHE implementation based on GPGPU, there are still many chances to further improve the performance with an acceptable overhead, such as shifting to the platforms with multiple GPGPUs.

2) Execution Time Breakdown: We calculate the kernel-level and operation-level execution time breakdown for executing the real workload. The results are as shown in Table 12 and Table 13, respectively. As shown in Figure 12, the NTT kernels take the largest proportion of the execution time for all workloads, which is up to 92.8% in LR. Considering the granularity of the operations, HROTATE becomes the most time-consuming operation, which is frequently used and contains a mass of NTT kernels.

D. Energy

To evaluate the energy efficiency of TensorFHE, we use NVIDIA-SMI to monitor the run-time GPGPU power, which is stable at 264 watts in our experiments due to the high hardware utilization. The energy efficiency of different CKKS operations is as shown in Table XI, which is up to 81.30 OPs/W in HADD. Besides, we also report the energy efficiency of various workload, which varies from 111.3 J/iteration (Packed Bootstrap) to 1320 J/iteration (ResNet-20). Table XI also shows that the energy consumption of TensorFHE is higher than the ASICs, due to the higher power of GPGPU.

E. Sensitivity Study to the Batch Size

Batch Size refers to the number of identical FHE operations simultaneously executed. As discussed in Section IV, the twiddle factor matrix can be shared by all NTT kernels with the same N and q. Therefore, a larger batch size allows more operations to fully utilize the reusable data and the hardware resources, which helps to improve the performance of the real workload based on FHE schemes. However, as the batched operations increase, the requirement for the VRAM capacity also grows, which is caused by the increasing intermediate data. Therefore, the batch size of TensorFHE is mainly determined by the VRAM capacity of the GPGPU.

Figure 14 presents the impact of the batch size on the overall performance of TensorFHE. Here we use the Default configuration as shown in Table V, and only change the value of the batch size (denoted as BS). As the BS increases from 32 to 1024, the performance of all kernels is improved. For example, the ForbeniusMap achieves the best performance improvement with BS=1024, which achieves 31.4% performance improvement compared with BS=128. However, Figure 14 also shows that different kernels have various optimal BS values. To balance the workload of different kernels, in this paper, we take $BS = 128$ as the default configuration.

TABLE IX
GPGPU OCCUPANCY OF THE TENSORFHE OPERATIONS.

	HMULT	HROATE	RESCALE	HADD	CMULT
Occupancy	90.3%	90.1%	88.9%	85.3%	88.1%

TABLE X
PERFORMANCE COMPARISON FOR THE FULL FHE WORKLOADS. TAKING THE EXECUTION TIME (IN SECONDS) AS THE METRIC.

	ResNet-20	LR	LSTM	Packed Bootstrapping
CPU [58]	88320	22784	27488	550.4
F1+ [57]	172.3	40.9	82.3	1.8
CraterLake [58]	15.9	7.6	4.4	0.1
BTS [38]	122.2	1.8	-	-
ARK [35]	18.8	0.49	-	-
100x* [33]	602.9	49.6	-	36.9
TensorFHE	316.1	14.1	123.1	13.5

*The execution time of ResNet-20 and Packed-Bootstrapping of 100x are estimated based on the executed numbers of HE operands.

F. Sensitivity Study to the Polynomial Length

The polynomial length (denoted as N) of TensorFHE refers to the number of coefficients in the polynomials used to represent the input data. A larger N means that the input data will be represented as a longer polynomial with a higher security level guarantee [31]. However, it also involves a higher computational overhead. Here we fix other attributes and vary the value of N from 65536 to 2048. Figure 15 shows the performance, measured as the execution time.

We can see that TensorFHE with $N = 65536$ performs considerably worse than other configurations. This is because a longer polynomial increases the computational workloads for all kernels, especially for the NTT kernels, which also require a much larger twiddle factor matrix. On the other hand, TensorFHE runs much faster with smaller N. Especially, as the N decreases from 65536 to 2048, the NTT kernel gains 20.6× speedup, since the computational workload is reduced by 97%. However, this also decreases the security level. In

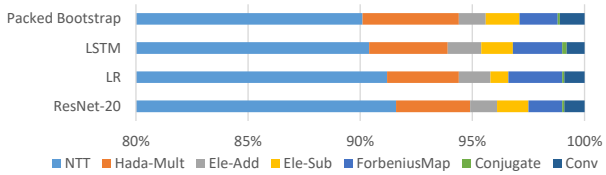

Fig. 12. Kernel-level Execution Time Breakdown

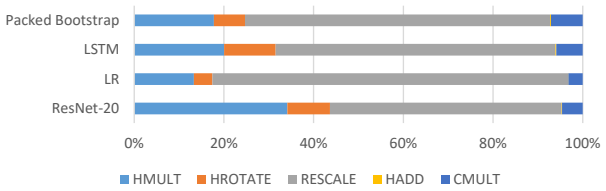

Fig. 13. Operation-level Execution Time Breakdown

this paper, we use a default configuration with $n = 65536$ to provide the acceptable security level for the real workloads.

VII. Discussion

Further Performance Improvement. Although the current TensorFHE achieves significant performance improvement, there are still several opportunities for further optimization, which can be mainly summarized in two aspects: on the one hand, the arithmetic kernels can be further optimized by improving the parallelism or using a streaming pipeline; on the other hand, extending TensorFHE to the platform with multiple GPGPUs would help to increase the batch size, which improves the performance of complex workloads by further improving the throughput of CKKS operations. We will implement these characteristics in future TensorFHE.

Generality. In this paper, we focus on the emerging CKKS scheme. However, our proposed TensorFHE can also support other FHE schemes, such as BFV [26] and BGV [7]. For a new FHE scheme, the *API Layer* provides new APIs to the user applications and invokes the required kernels, while the *Kernel Layer* would require new arithmetic kernels to support different algorithms.

Security Vulnerability. TensorFHE achieves significant performance improvement on GPGPU without changing the key algorithm of CKKS. Therefore, though the GPGPU might be untrusted, security can still be guaranteed since all the computations are fulfilled on the encrypted data.

VIII. Related Work

GPU-based FHE acceleration. Several previous works have been proposed to accelerate various FHE schemes on GPGPU [1], [4], [23], [33], [61]. Privft [1] the first CKKS implementation with the RNS-variant on GPGPU. It directly maps the original CKKS algorithm to the GPGPU using the data tiling. However, it failed to support bootstrap and provided no deep insights. Based on this work, Ref [61] uses the Barret Reduction method [39] and CRT to improve the performance of FHE schemes on GPGPU. 100x [33] is the first CKKS implementation on GPGPU with bootstrap support. It analyzes the bottleneck of memory accesses and then improves the CKKS performance using the optimized

TABLE XI
Energy Efficiency of TensorFHE

Energy Efficiency of CKKS operations (OPs/W)				
HMULT	HROATE	RESCALE	HADD	CMULT
0.57	0.57	66.67	81.30	66.67

Energy Consumption of full workload (J/iteration)				
	ResNet-20	LR	LSTM	Packed Bootstrap
ARK [35]	32.5	19.8	-	-
CraterLake [58]	79.7	38.1	44.2	1.3
TensorFHE	1320	58.27	1015.3	111.3

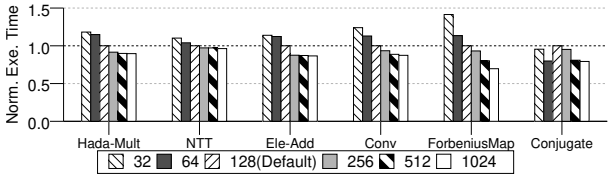

Fig. 14. Impact of batch size on execution time.

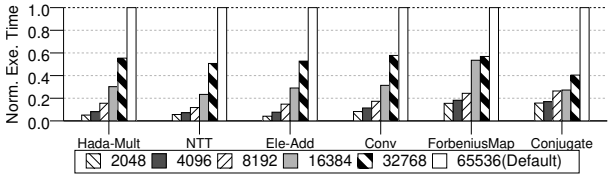

Fig. 15. Sensitivity to *polynomial length* of TensorFHE.

function kernel with fewer memory accesses. Ref [4] accelerates the FV scheme on GPGPU, which uses the CRT and discrete Galois transformation [2] to avoid the additional multi-precision arithmetic operations. Ref [23] improves the performance of the BGV scheme on GPGPU, which optimizes the modular operations with the Barret Reduction method [39] and overlaps the data transfer between the host and GPU with the computation. These previous works focus on the optimization of memory accessing, while our TensorFHE makes an effort to improve the performance of CKKS by enhancing hardware efficiency and utilization with algorithm optimizations.

Hardware-based FHE accelerator. Previous works also explore to accelerate FHE schemes by using the FPGA [20], [21], [56], [60] or ASIC accelerators [35], [38], [57], [58]. Due to the limited on-chip hardware resources, the FPGA-based approaches focus on the acceleration for the specific key operations, such as the NTT/INTT [9], [56], the ModMult [10], [56], and the ModAdd [9]. However, for the real workloads, the overall performance suffers from the frequently data transfer caused by the partial acceleration support. On the other hand, ASIC accelerators support all FHE operations and significantly improve the performance of FHE-based workloads [38]. But, all of these works require the huge on-chip storage (i.e., 256MB for F1+ [57] and CraterLake [58], 512MB for BTS [38] and ARK [35]), which leads to the over-high implementation cost. Our proposed TensorFHE proves that, with algorithm optimization based on the understanding of micro-architectural characteristics, it is possible to achieve comparable performance for the FHE workloads on GPGPUs.

IX. Conclusion

In this paper, we provide a detailed analysis of the inefficiency of running FHE operations on GPGPU and then propose TensorFHE, which is a pure software FHE acceleration solution based on single GPGPU. TensorFHE uses a hierarchical model to reconstruct the CKKS, which decomposes the FHE operations into a series of reusable kernels. Then, TensorFHE optimizes the algorithm of the NTT kernel to fit it with the emerging TCUs and utilizes the regular CUDA cores to accelerate the other kernels. Moreover, to fully utilize the potential data parallelism of GPGPU, TensorFHE uses an operation-level batching technique to allow more FHE operations to be executed simultaneously. In this way, the proposed TensorFHE significantly improves the performance of FHE applications by executing more FHE operations in the certain period of time. The experiment results show that TensorFHE achieves much higher performance than all state-of-the-art FHE acceleration solutions on CPU, GPU and FPGA. More surprisingly, TensorFHE provides comparable performance when compared with the state-of-the-art ASIC FHE accelerators, and even achieves higher performance in some applications. Considering the high expense of ASIC implementation, we believe that TensorFHE can be a competitive candidate for applying FHE to the cloud computing servers in the near future, and will possibly inspire more works.

References

[1] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226 544–226 556, 2020.

[2] A. Al Badawi, B. Veeravalli, and K. M. M. Aung, "Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution," in *Future of Information and Communication Conference*. Springer, 2018, pp. 666–682.

[3] A. Al Badawi, B. Veeravalli, and K. M. M. Aung, "Faster number theoretic transform on graphics processors for ring learning with errors based cryptography," in *2018 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*. IEEE, 2018, pp. 26–31.

[4] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.

[5] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full rns variant of fv like somewhat homomorphic encryption schemes," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 423–442.

[6] 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)*, 2016, pp. 423–442.

[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[8] C. Brunelli, H. Berg, and D. Guevorkian, "Approximating sine functions using variable-precision taylor polynomials," in *2009 IEEE Workshop on Signal Processing Systems*. IEEE, 2009, pp. 057–062.

[9] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High-speed fully homomorphic encryption over the integers," in *International Conference on Financial Cryptography and Data Security*. Springer, 2014, pp. 169–180.

[10] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *Cryptology ePrint Archive*, 2013.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[12] H. Chen and K. Han, "Homomorphic lower digits removal and improved fhe bootstrapping," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 315–337.

[13] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "Cfntt: Scalable radix-2/4 ntt multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 94–126, 2022.

[14] J. H. Cheon, K. Han, and M. Hhan, "Faster homomorphic discrete fourier transforms and improved fhe bootstrapping." *IACR Cryptology ePrint Archive*, vol. 2018, p. 1073, 2018.

[15] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *International Conference on Selected Areas in Cryptography*. Springer, 2018, pp. 347–368.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International conference on the theory and application of cryptology and information security*. Springer, 2017, pp. 409–437.

[17] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International conference on the theory and application of cryptology and information security*. Springer, 2017, pp. 409–437.

[18] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch, "What is the fast fourier transform?" *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, 1967.

[19] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[20] D. B. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An fpga co-processor implementation of homomorphic encryption," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[21] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an fpga-accelerated homomorphic encryption co-processor," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 193–206, 2016.

[22] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[23] J. Dong, "Accelerating bgv scheme of fully homomorphic encryption using gpus," Ph.D. dissertation, WORCESTER POLYTECHNIC INSTITUTE, 2016.

[24] S. Durrani, M. S. Chughtai, M. Hidayetoglu, R. Tahir, A. Dakkak, L. Rauchwerger, F. Zaffar, and W.-m. Hwu, "Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 345–355.

[25] S. Erabelli, "pyfhe-a python library for fully homomorphic encryption," Ph.D. dissertation, Massachusetts Institute of Technology, 2020.

[26] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[27] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[28] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[29] A. Gupta and K. R. Rao, "A fast recursive algorithm for the discrete sine transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, no. 3, pp. 553–557, 1990.

[30] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale." in *AAAI Conference on Artificial Intelligence (AAAI)*, 2019, pp. 9466–9471.

[31] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.

[32] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[33] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[34] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated gpu modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 473–486.

[35] J. Kim, G. Lee, S. Kim, G. Sohn, J. Kim, M. Rhu, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," *arXiv preprint arXiv:2205.00922*, 2022.

[36] M. Kim and K. Lauter, "Private genome analysis through homomorphic encryption," in *BMC medical informatics and decision making*, vol. 15, no. 5. BioMed Central, 2015, pp. 1–12.

[37] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 264–275.

[38] 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*, 2022, pp. 711–725.

[39] M. Knezevic, F. Vercauteren, and I. Verbauwhede, "Faster interleaved modular multiplication based on barrett and montgomery reduction methods," *IEEE Transactions on Computers*, vol. 59, no. 12, pp. 1715–1721, 2010.

[40] J. Konečnỳ, B. McMahan, and D. Ramage, "Federated optimization: Distributed optimization beyond the datacenter," *arXiv preprint arXiv:1511.03575*, 2015.

[41] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions." in *International Conference on Machine Learning (ICML)*, 2022, pp. 12 403–12 422.

[42] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network." *ACM Transactions on Accessible Computing (TACCESS)*, vol. 10, pp. 30 039–30 054, 2022.

[43] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *International Conference on Cryptology and Network Security*. Springer, 2016, pp. 124–139.

[44] P. E. Madrid, B. Millar, and E. E. Swartzlander, "Modified booth algorithm for high radix fixed-point multiplication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 2, pp. 164–167, 1993.

[45] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2019, pp. 253–260.

[46] C. Mouchet, J.-P. Bossuat, J. Troncoso-Pastoriza, and J. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[47] T. Mukherjee, *Cyclotomic polynomials in Ring-LWE homomorphic encryption schemes*. Rochester Institute of Technology, 2016.

[48] NVIDIA, "cuda 11.0," https://developer.nvidia.com/cuda-11-0-0-download-archive,2022.

[49] NVIDIA, "Cutlass 2.8,," https://github.com/NVIDIA/cutlass,2022.

[50] NVIDIA, "Nvidia tesla v100 gpu architecture," *Tesla NVIDIA*, 2017.

[51] NVIDIA, "Nvidia a100 tensor core gpu architecture," *NVIDIA*, 2022.

[52] NVIDIA, "Nvidia nsight systems," *NVIDIA*, 2022.

[53] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[54] R. Podschwadt and D. Takabi, "Classification of encrypted word embeddings using recurrent neural networks." in *Web Search and Data Mining (WSDM)*, 2020, pp. 27–31.

[55] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *International conference on cryptology and information security in Latin America*. Springer, 2012, pp. 139–158.

[56] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[57] 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 *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[58] 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 *ISCA*, 2022, pp. 173–187.

[59] V. Shoup, "A new polynomial factorization algorithm and its implementation," *Journal of Symbolic Computation*, vol. 20, no. 4, pp. 363–397, 1995.

[60] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.

[61] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using gpu," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2014, pp. 2800–2803.

[62] E. W. Weisstein, "Fermat's little theorem," *https://mathworld. wolfram. com/*, 2004.

[63] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino, "Single-database private information retrieval from fully homomorphic encryption," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 5, pp. 1125–1134, 2012.