CHAM: A Customized Homomorphic Encryption Accelerator for Fast Matrix-Vector Product

Xuanle Ren*, Zhaohui Chen*†, Zhen Gu*‡, Yanheng Lu*, Ruiguang Zhong*, Wen-jie Lu§, Jiansong Zhang*,
Yichi Zhang*, Hanghang Wu¶, Xiaofu Zheng¶, Heng Liu*, Tingqiang Chu¶,
Cheng Hong§, Changzheng Wei¶, Dimin Niu*and Yuan Xie*

* DAMO Academy, Alibaba Group, China
{xuanle.rxl, chenzhaohui.czh, guzhen.gz, yanheng.lyh, ruiguang.zrg, jiansong.zjs, yichi.zyc, hengliu.lh, dimin.niu, y.xie}@alibaba-inc.com
† Peking University, China ‡ Tsinghua University, China § Ant Group, China {juhou.lwj, vince.hc}@antgroup.com
¶ AntChain, Ant Group, China {hanghang.whh, linke.zxf, chutingqiang.ctq, changzheng.wcz}@antgroup.com

***Abstract*—Homomorphic encryption (HE) is a promising technique for privacy-preserving computing because it allows computation on encrypted data without decryption. HE, however, suffers from poor performance due to enlarged data size and exploded amount of computation. Related work has been proposed to accelerate HE using GPUs, FPGAs, and ASICs. The existing work, however, aims at specific HE schemes and fails to consider the fast-evolving algorithms. For example, HE algorithms that combine different HE schemes have demonstrated capability of supporting more types of HE operations and ciphertexts. Moreover, some existing hardware accelerators target small HE operations (such as number theoretic transform and key-switch), which however provides limited or even neglected performance improvement for end-to-end applications. To better support existing privacy-preserving applications (*e.g.*, logistic regression and neural network inference), we propose *CHAM*, an HE accelerator, for high-performance matrix-vector product, which can be easily extended to 2-D and 3-D convolutions. Motivated by the evolution of algorithms, *CHAM* supports not only traditional HE operations, but also different types of ciphertexts and the conversion between them. We implement *CHAM* with Xilinx FPGAs. The evaluation demonstrates $1800\times$ speed-up for matrix-vector product, $36\times$ speed-up for logistic regression, and $144\times$ speed-up for Beaver triple generation compared to the existing work.**

***Index Terms*—homomorphic encryption, accelerator, matrix-vector product, logistic regression**

I. Introduction

Privacy-preserving computing is a technique that enables multiple parties to collaboratively learn a shared prediction model with exchanging minimal amounts of data. This goal can be achieved using protocols such as secure multi-party computation (MPC) [28] and federated learning [16]. Among these protocols, homomorphic encryption (HE), capable of computing on encrypted data, is used for exchanging data between parties.

Fully homomorphic encryption (FHE) scheme, which is able to evaluate arbitrary-depth functions, was successfully constructed by Craig Gentry [15]. Since then, efficient HE schemes based on the learning-with-errors (LWE) problem and its ring variant (RLWE) have been proposed due to their simplicity in decryption and arithmetic features, including Brakerski/Fan-Vercauteren (B/FV) [12], Cheon-Kim-Kim-Song (CKKS) [8], and fully homomorphic encryption over the torus (TFHE) [9].

Nevertheless, FHE is considered far from wide application because of its poor performance. Most mainstream HE schemes suffer from an explosion of both ciphertext size ($\times 10^2$ to $\times 10^5$) and computation ($\times 10^3$ to $\times 10^6$). For example, inference of an encrypted neural network consumes 250 seconds [11], [17]. This problem can be mitigated using batched processing [3], [8], [12]. For example, up to 4096 encrypted images can be evaluated simultaneously such that the cost of a single image is amortized [3]. The performance of FHE becomes even worse when supporting computation with arbitrary depth. More precisely, a ciphertext, upon encrypted, is associated with a noise budget that will be consumed during computation. To ensure correct decryption, this noise budget should not be exhausted, meaning that only limited-depth computation is allowed. Bootstrapping overcomes this limitation by refreshing the noise budget before it is about to exhaust [15]. However, bootstrapping introduces a huge amount of computation which usually dominates the whole computation [32].

Customized, highly-parallel hardware (*e.g.*, FPGAs, GPUs, and ASICs) is used for accelerating FHE, especially for bootstrapping. The FPGA-based approach targets HE operators, such as polynomial multiplication and number theoretic transform (NTT) [1], [24], [25], [27], [31], [33]. A representative work accelerates key-switch by implementing a pipelined architecture on FPGA [31]. However, accelerating solely key-switch provides limited performance gain from the perspective of high-level applications. GPUs demonstrate tens of times of speed-up compared to CPUs [2], [10], [19], [20]. The bottleneck of GPU, however, resides in limited shared memory that cannot accommodate large polynomials and the intermediate results during HE evaluation. ASICs achieve tremendous performance improvement, benefiting from the use of high-frequency clock, high-bandwidth memory, and dense compute logic [13], [14], [22], [23], [32]. The chip area of these ASICs, however, is extremely large ($100\,\mathrm{mm}^2 \sim 400\,\mathrm{mm}^2$).

Compared to using bootstrapping, a more practical solution is to avoid deep homomorphic evaluation by combining HE with other techniques. For example, inference of an encrypted neural network can be realized using HE and garbled circuits [21]. In particular, only linear layers are evaluated homomorphically while the non-linear layers are handled by garbled circuits. Since linear operations are usually quite shallow, the encryption parameter required by them is much smaller than the case where bootstrapping is supported ($N = 2^{12}$ vs. $N = 2^{16}$). With this protocol, inference of ResNet-20 becomes 1000 times faster than the HE-only solutions [3]. More performance gain can be achieved if implemented with ASICs [30].

Besides the overhead caused by bootstrapping, HE also falls short of supporting different types of functions. In particular, the commonly used HE schemes (*i.e.*, B/FV and CKKS) cannot support non-linear functions efficiently (*e.g.*, activation function in neural networks). A solution is to approximate these functions using high-order polynomials [17], but it causes an already-trained model to be modified and even degradation of model accuracy. For example, we observe that the LeNet-5 has been modified to many homomorphic variants [5], [11], [17], some of which demonstrate an obvious drop of accuracy. To address this problem, two groups of novel algorithms have been proposed. First, the efficiency of HE computation can benefit from using multiple types of ciphertexts that can be converted to each

other [7]. Second, different HE schemes (*i.e.*, B/FV, CKKS, and TFHE) may compose a hybrid scheme that supports both linear and non-linear functions effectively without introducing approximation error [4], [26]. These new HE algorithms, however, have not been implemented on FPGA or ASIC.

In this work, we propose *CHAM*, a Customized Homomorphic encryption Accelerator for high-performance homomorphic Matrix-vector product (HMVP), with algorithm-architecture co-design. First, based on the observation that HE is suitable to and widely used for computing linear functions, *CHAM* targets a novel algorithm of homomorphic MVP. In particular, *CHAM* supports different types of ciphertexts (*i.e.*, RLWE and LWE) and the conversion between them, which demonstrates more flexibility than conventional algorithms. Second, *CHAM* employs a customized, fully-pipelined architecture that can effectively utilize parallelism of hardware. For NTT, we propose a novel, compute-efficient architecture that enables pipelined data flow without any bubble. *CHAM* has been deployed within a privacy-preserving machine solution in one of the top cloud service providers. To the best of our knowledge, *CHAM* is the first HE accelerator deployed for commercial applications. The evaluation results on benchmarks demonstrate $1800\times$ speed-up for HMVP, $36\times$ speed-up for logistic regression, and $144\times$ speed-up for Beaver triple generation compared to existing work. We provide an open-source release of *CHAM* (https://github.com/alibaba-damo-academy/damo_ctl_cham).

In the rest of this paper, we first provide background for HE in Section II. Next, we describe the architecture design of *CHAM* in Section III and IV. Finally, in Section V, we present implementation of *CHAM* and performance evaluation for typical benchmarks.

II. Background

A. Notations

This section introduces some basic notations and HE-related terminologies. Raw data in the real world is called *cleartext* and it is encoded to the so called *plaintext*. Moreover, some data should be protected via encryption, and its plaintext is later encrypted into the so called *ciphertext*.

RLWE is an encryption scheme commonly used in HE schemes (*e.g.*, B/FV and CKKS). In *CHAM*, the plaintext is encoded as polynomials and the ciphertext is encrypted as a tuple of two polynomials, written as $(b(X), a(X))$. Throughout this paper, the plaintext encoding the cleartext a is denoted by $\mathtt{pt}^{(a)}$, and similarly, the ciphertext encrypting a is denoted by $\mathtt{ct}^{(a)}$.

B. Homomorphic Matrix-Vector Product (HMVP)

In applications like *logistic regressions* [16] and *Beaver triple generation* [28], the computation is mainly composed of multiple matrix-vector products (MVP). In privacy-preserving applications, the MVPs are computed homomorphically, namely homomorphic matrix-vector product (HMVP). In such cases, the matrix A is encoded a set of plaintexts $\{\mathtt{pt}^{(A_i)}\}$ and the vector $\vec{v}$ is encrypted as a ciphertext $\mathtt{ct}^{(\vec{v})}$. After computing HMVP, $A \cdot \vec{v}$ is encrypted in a new ciphertext $\mathtt{ct}^{(\vec{u})}$ as demonstrated in Alg. 1.

C. Dot Products via Polynomial Multiplication

The plaintext encoding the i-th row A_i and the vector $\vec{v}$ are

$$\mathtt{pt}^{(A_i)} = A_{i,0} - \sum_{j=1}^{N-1} A_{i,j} X^{N-j}, \quad \mathtt{pt}^{(\vec{v})} = \sum_{j=0}^{N-1} \vec{v}_j X^j, \quad (1)$$

respectively. Then the homomorphic product of $\mathtt{pt}^{(A_i)}$ and $\mathtt{ct}^{\vec{v}}$ is a ciphertext with its plaintext being

$$\mathtt{pt}^{(A_i)} \times \mathtt{pt}^{(\vec{v})} = \sum_{j=0}^{N-1} \Big(\sum_{k=0}^{N-1-j} A_{i,k} \vec{v}_{k+j} - \sum_{k=N-j}^{N-1} A_{i,k} \vec{v}_{k+j-N} \Big) X^j, \quad (2)$$

whose constant coefficient is actually the inner product of vectors A_i and $\vec{v}$. After computing all the homomorphic products of $\mathtt{pt}^{(A_i)}$ and $\mathtt{ct}^{(\vec{v})}$, we obtain a set of ciphertexts $\mathtt{ct}_i$ encrypting the inner products $A_i \cdot \vec{v}$ in the constant coefficients of their plaintexts.

D. Packing Encrypted Scalars

For a m-row matrix A, we obtain m ciphertexts rather than a single ciphertext after HMVP. Only the constant coefficients of the plaintexts are useful. Hence, unpacking the constant coefficients from the m ciphertexts and repacking them together into a single ciphertext is required in *CHAM*. For the purpose of unpacking and repacking, we apply the method proposed by Hao Chen *et al.* [7]. Generally speaking, we extract all the constant coefficients of the m plaintexts encrypted in ciphertexts, and then cast then as new ciphertexts, which eventually would be packed in a result ciphertext. PackTwoLWEs presented in Alg. 2 packs two ciphertexts and is recursively called to pack multiple ciphertexts by PackLWEs as shown in Alg. 3. Actually, to pack two ciphertexts $\mathtt{ct}^{(i)}$, a PackTwoLWEs procedure consists of multiplying a monomial, subtraction, addition, Automorphism, and KeySwitch with its input ciphertexts being preprocessed by a procedure ExtractLWEs, which convert ciphertexts $\mathtt{ct}^{(i)} = (\mathbf{u}^{(i)}(X), \mathbf{v}^{(i)}(X))$ in the following manner:

$$\text{ExtractLWEs}(\mathtt{ct}^{(i)}) = (\mathbf{u}_0^{(i)}, \mathbf{v}_0^{(i)} - \sum_{j=1}^{N-1} \mathbf{v}_j^{(i)} X^{N-j}) \quad (3)$$

Algorithm 1 Coefficient-encoded matrix-vector product

Input: A matrix $A^{(m\times N)} \coloneqq \{A_i\}_{i=0,\dots,m-1}$ and a vector $\vec{v}$
Output: A vector $\vec{u} = A \cdot \vec{v}$
1: **for** $i \leftarrow 0$ to $m-1$ **do**
2: $\quad \mathtt{ct}^{(i)} = \mathtt{pt}^{(A_i)} \times \mathtt{ct}^{(\vec{v})}$ ▷ Dot product
3: $\quad \mathtt{ct}_i = \text{ExtractLWEs}(\mathtt{ct}^{(i)})$ ▷ Extract coefficient
4: **end for**
5: $\mathtt{ct}^{(\vec{u})} = \text{PackLWEs}(\mathtt{ct}_0, ..., \mathtt{ct}_{m-1})$ ▷ Pack coefficients

Algorithm 2 PackTwoLWEs

Input: An automorphism index l, two RLWE ciphertexts $\mathtt{ct}_i = \text{ExtractLWEs}(\mathtt{ct}^{(i)}) \quad i \in \{0,1\}$. An index-$l$ Keyswitch key KSK_l
Output: An RLWE ciphertext $\mathtt{ct}$
1: $\mathtt{ct}_{mono} = X^{N/2} \cdot \mathtt{ct}_1$ ▷ Multiply a monomial
2: $(\mathbf{b}_+(X), \mathbf{a}_+(X)) = \mathtt{ct}_0 + \mathtt{ct}_{mono}$
3: $(\mathbf{b}_-(X), \mathbf{a}_-(X)) = \mathtt{ct}_0 - \mathtt{ct}_{mono}$
4: $(\mathbf{b}_\mathbf{A}(X), \mathbf{a}_\mathbf{A}(X)) = (\mathbf{b}_-(X^{2l+1}), \mathbf{a}_-(X^{2l+1}))$ ▷ Automorphism
5: **return** $(\mathbf{b}_+(X) + \mathbf{b}_\mathbf{A}(X), \mathbf{a}_+(X)) + \mathbf{a}_\mathbf{A}(X) \cdot \text{KSK}_l$ ▷ KeySwitch

Algorithm 3 PackLWEs

Input: RLWE ciphertexts $\{\mathtt{ct}_i\}_{i=0}^{2^l-1}$
Output: An RLWE ciphertext $\mathtt{ct}$
1: **if** $l = 0$ **then**
2: $\quad$ **return** $\mathtt{ct}_0$
3: **else**
4: $\quad \mathtt{ct}_{even} = \text{PackLWEs}(\{\mathtt{ct}_{2i}\}_{i=0}^{2^{l-1}-1})$
5: $\quad \mathtt{ct}_{odd} = \text{PackLWEs}(\{\mathtt{ct}_{2i+1}\}_{i=0}^{2^{l-1}-1})$
6: $\quad$ **return** $\text{PackTwoLWEs}(l, \{\mathtt{ct}_{even}, \mathtt{ct}_{odd}\})$
7: **end if**

E. Related Work

Besides encoding cleartexts as coefficients of the plaintexts directly (*i.e.*, *coefficient-encoding*), we can also encode cleartext using a single-instruction multiple-data (SIMD) method that can carry out the same computation over a large number of *slots* using only one ciphertext (called *batch-encoding*). In many HMVP frameworks [21],

a plaintext matrix A is encoded with batching, and it can be directly rotated and summed up to obtain the result ciphertext. Compared to batch-encoded HMVP [21], our coefficient-encoded HMVP (Alg. 1) reduces computation complexity from $\mathcal{O}(m \log_2 N)$ to $\mathcal{O}(m)$. When compared to the diagonal-encoded method proposed in [21] whose complexity is also $\mathcal{O}(m)$, Alg. 1 is still faster because coefficient-encoding incurs much smaller overhead. Moreover, Alg. 1 can be extended to other linear functions, such as 2-D and 3-D convolutions through encoding the original tensors in similar ways [18].

F. Security Model and Parameter Selection

In this work, we adopt a two-party computation model which is widely adopted in MPC and federated learning. More precisely, party $\mathcal{A}$ owns a share of a vector and party $\mathcal{B}$ owns the other share of the vector as well as a matrix. $\mathcal{A}$ encrypts her vector share and sends it to $\mathcal{B}$. $\mathcal{B}$ then combines two vector shares and multiplies it to the matrix. We assume that $\mathcal{B}$ is a semi-honest adversary. In other words, while $\mathcal{B}$ tries to learn as much as possible from the data provided by $\mathcal{A}$, $\mathcal{B}$ honestly follows the prescribed protocol.

The selection of encryption parameters (*i.e.*, polynomial degree N and moduli q_i) is based on the required security level and plaintext space. In particular, a high security level and/or a large plaintext space demand large encryption parameters. In this work, we select $N = 4096$, which is sufficient to support linear homomorphic computation. This corresponds to a space of $109\,\text{bit}$ where $70\,\text{bit}$ (corresponding to two $35\,\text{bit}$ moduli) is used for representing plaintext and ciphertext, while the other $39\,\text{bit}$ is used as a special modulus for key-switching. Thus, a ciphertext consists of four 4096-degree polynomials, while a plaintext consists of two polynomials. If *augmented* with the special modulus, they consists of six and three polynomials, respectively.

III. Proposed Architecture Design

In this section, we describe the architecture of *CHAM* and how it is achieved through searching the design space.

A. Architecture Overview

CHAM aims to accelerate coefficient-encoded matrix-vector product described in Alg. 1. As shown in Fig. 1a, *CHAM* consists of a number of compute engines, each of which employs a fully-pipelined architecture[1] to improve resource utilization and maximize throughput. All the polynomials within a plaintext and a ciphertext are processed in parallel, each corresponding to a functional unit (*e.g.*, NTT, MULTPOLY, and INTT).

The DOTPRODUCT module takes augmented plaintext and ciphertext as input. The input polynomials are transformed from *coefficient-domain* to *NTT-domain* (stage-1) so the multiplication of polynomials can be simplified from a convolution to coefficient-wise multiplication (stage-2). The result is then transformed back to coefficient-domain (stage-3). The stage-4 rescales the augmented ciphertext to normal format through dividing the ciphertext by the $39\,\text{bit}$ special modulus. The purpose of this stage is to reduce the noise introduced by polynomial multiplication (from $30\,\text{bit}$ to $26\,\text{bit}$). The EXTRACTLWES module simply extracts the dot-product result from the RLWE ciphertext and saves it as an LWE ciphertext. It resides in the same stage with the RESCALE unit (stage-4) because both of their execution is coefficient-wise over polynomials and therefore easy to be combined.

The PACKTWOLWES module, corresponding to Alg. 2, packs the dot-product results (appearing as LWE ciphertexts) to a single RLWE ciphertext (stage-5∼9). Note that the packing procedure in Alg. 3 can be described as a binary tree that takes inputs from the leaf nodes and outputs the result in the root node. Totally 4095 reductions are required to pack 4096 ciphertexts since the PACKTWOLWES module reduces two ciphertexts into one each time. The intermediate reduction results are stored in a reduce buffer. Once the intermediate reduction results are ready for the next-level reduction, they preempt the pipeline and stalls the execution of the preceding stages.

[1]The pipeline mentioned here refers to a *macro-pipeline*, meaning that each stage of it contains multiple functional units and execution of a stage takes thousands of clock cycles.

B. Design Space Exploration

According to the roofline model evaluated on the Xilinx U200 FPGA in Fig. 2a, we clearly observe that the compute intensity of HE operations (*e.g.*, NTT and key-switch) is much smaller than HMVP. Invoking these HE operations individually will cause intensive memory access and therefore degrade overall performance. Thus, in order to achieve the best performance, *CHAM* adopts a fully-customized architecture to accelerate HMVP as a whole, instead of individual HE operations (*e.g.*, NTT and key-switch).

Next, we explore the design space for implementing HMVP, including 1) how to split the pipeline, 2) how to choose the number of compute engines, modules and FUs, 3) how to decide the parallelism of each FU, and 4) the size of buffers. We plot the results in Fig. 2b with each design choice positioned by its performance and resource utilization. The optimal choices reside in two points, i.e., (9-stages, 1×PACKTWOLWES, 6×NTT, 4-PE NTT, 2×compute engines) and (9-stages, 1×PACKTWOLWES, 6×NTT, 8-PE NTT, 1×compute engine). *CHAM* corresponds to the first point. Note that for each design choice in Fig. 2b, we have to ensure that all pipeline stages have similar latency and throughput in order to maximize the overall performance. For example, if stage-A involves k-times more computation than stage-B, then the parallelism of stage-A should be k-times larger than the parallelism of stage-B (*i.e.*, $P_A = kP_B$).

C. Heterogeneous System Design

CHAM is implemented in a heterogeneous system of CPU and FPGA. We maximize its performance by interleaving computation and data transfer between the FPGA and the host CPU. Fig. 1b illustrates how it works in case of different numbers of CPU threads and *CHAM* compute engines. On the host-side, we pipeline data transfer and computation using multiple threads. On the FPGA-side, we use RAMs to buffer the input and output data of each thread.

A software stack with runtime and driver are developed to support high-level application. In addition to provide APIs for application, the runtime also support reliability, availability, and serviceability (RAS) features including FPGA register loading error handling, FPGA hang/reset, and FPGA health monitoring.

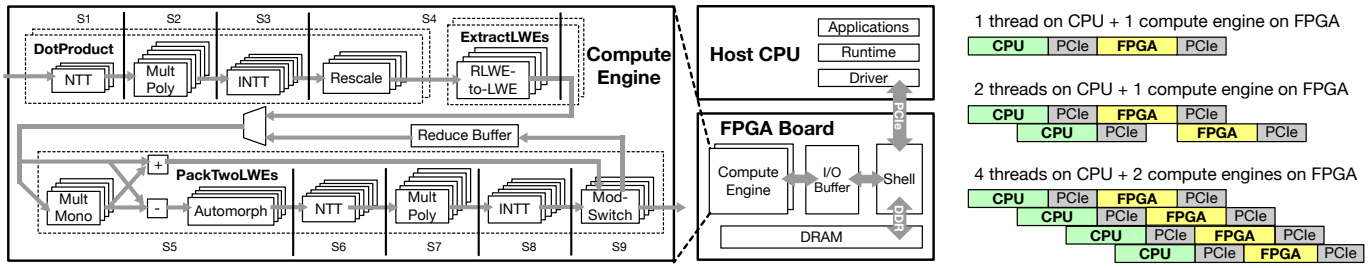
IV. Microarchitecture

In this section, we elaborate on the design of FUs, including number theoretic transform and polynomial processing units.

Algorithm 4 Constant-geometry forward NTT

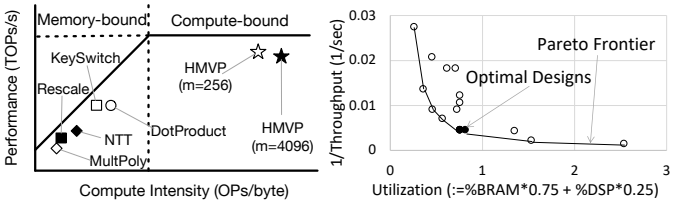
Input: Polynomial $\mathbf{a}(X)$, twiddle factors $\omega_{2N}[\log_2 N * N/2]$.
Output: $\bar{\mathbf{a}}(X) = \text{NTT}(\mathbf{a}(X))$ in bit-reversed order

for $i \leftarrow 0$ to $\log_2 N$ **do** ▷ i is stage index
  for $j \leftarrow 0$ to $N/2$ **do** ▷ j is butterfly index
    $\omega_{ij} = \omega[i \cdot N/2 + j]$ ▷ Fetch factor
    $\bar{\mathbf{a}}(X)_{2j} = \mathbf{a}(X)_j + \mathbf{a}(X)_{j+N/2} \cdot \omega_{ij}$ ▷ Butterfly
    $\bar{\mathbf{a}}(X)_{2j+1} = \mathbf{a}(X)_j - \mathbf{a}(X)_{j+N/2} \cdot \omega_{ij}$
  end for
  if $i \neq logN - 1$ **then**
    $\mathbf{a}(X) = \bar{\mathbf{a}}(X)$
  end if
end for



(a) (b)

Figure 1: (a) The architecture and system-view of *CHAM*. (b) An illustration for the pipelined execution of multi-thread CPU and FPGA.



(a) (b)

Figure 2: (a) The roofline model for *CHAM*, where an operation refers to a 27-by-18 integer multiplication because it fits a DSP slice on FPGA. (b) Various design points are explored so that the best performed designs that can fit the FPGA are selected.

A. Number Theoretic Transform (NTT)

NTT is a generalization of the discrete Fourier transform (DFT) to finite fields. The forward NTT and inverse NTT (INTT) indicate the conversion function between the normal polynomial form and NTT representation. In the NTT context, multiplying large polynomials $\mathbf{a}(X)$ and $\mathbf{b}(X)$ is performed by $\mathbf{c}(X) = \text{INTT}(\text{NTT}(\mathbf{a}(X)) \circ \text{NTT}(\mathbf{b}(X)))$, where "$\circ$" refers to coefficient-wise multiplication.

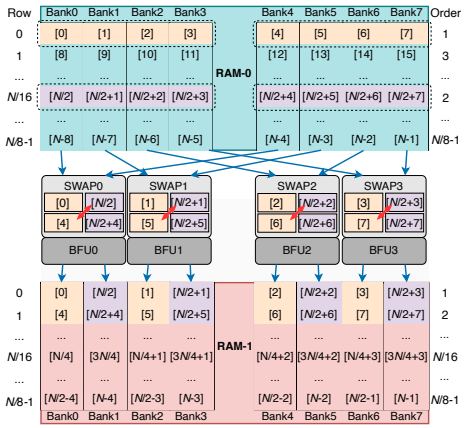


Figure 3: The NTT datapath for four BFUs.

Algorithm 4 shows a constant-geometry NTT [6], [29]. The outer loop divides the computation of NTT into $\log_2 N$ stages, while each stage consists of $N/2$ independent butterfly operations.

The butterfly parallelism degree (n_{bf}) affects both performance and hardware utilization. On the one hand, the $N/2$ butterfly operations in each NTT stage can be parallelized with $n_{bf} = 2^s$ butterfly units (BFUs). On the other hand, although a larger n_{bf} indicates higher parallelism, *CHAM* prefers fully utilized RAMs. On-chip memory depth constraints the upper bound n_{bf} because a polynomial needs to be stored in n_{bf} RAM banks to support high parallelism.

Previously ASIC-based NTT solution (*F1* [13]) requires ($\sqrt{N} \times \sqrt{N}$)-element memory block, which is not feasible for FPGA design. On the other hand, the FPGA-friendly design (*HEAX* [31]) with block RAM optimization requires a large number of look-up table (LUT)-based multiplexers to handle the stage-variant memory access pattern.

In our work, the following optimizations are proposed to address these drawbacks in previous work.

1) Parallel constant-geometry dataflow: In our design, a polynomial is stored in 8 *round-robin* RAM banks. NTT is executed in a ping-pong fashion as shown in Fig. 3. More precisely, the polynomial coefficients are read from RAM-0 and written to RAM-1 during the $2r$-th stages, while the coefficients are then read from RAM-1 and written to RAM-0 during the ($2r$+1)-th stages. Therefore, the NTT process requires $(N/2 \cdot \log_2 N)/n_{tf}$ clock cycles. In this work, we set $n_{bf} = 4$ therefore all 1R1W RAM banks can be processed parallelly.

The consecutive coefficients of a polynomial are stored across RAM banks (*i.e.*, the coefficients $[0] \sim [7]$ are stored in all RAM banks at address 0), such that they can be read and written simultaneously. The coefficients are read in an up-and-down order (*i.e.*, $[0] \sim [7]$, $[N/2] \sim [N/2+7]$, $[8] \sim [15]$, ..., $[N-8] \sim [N-1]$) and written in ascending order (*i.e.*, $[0] \sim [7]$, $[8] \sim [15]$, ..., $[N-8] \sim [N-1]$). This read-write fashion ensures a fixed datapath (called *constant geometry*) between the NTT units and the RAM banks. The SWAP unit reorders the coefficients read by a BFU, for the purpose of supporting the constant geometry. For example, the unit of SWAP-0 exchanges the positions of coefficients $[4]$ and $[N/2]$.

2) Generation and storage of twiddle factors: The NTT operation involves a total number of $N-1$ twiddle factors, as shown in Fig. 4 ($N = 32$, for example). *CHAM* aims to assign each BFU a separate ROM bank for storing the corresponding twiddle factors. To this end, the four twiddle factors in a column are assigned to four BFUs and used in the same clock cycle (*e.g.*, indices 4 to 7 in *stage*-2). The size of twiddle factors is equal to the size of a polynomial (*i.e.*, N). Moreover, multiple NTT units may share the same copy of twiddle factors, such that only two sets of twiddle factors are required by a compute engine, one for NTT and the other for INTT.

3) Customized optimization for modular reduction: Modular multiplication is the most important but complicated operation in HE. If a finite field is defined over a modulus with low-hamming weight, then modular arithmetic can be significantly simplified. Following the security model described in Section II-F, we choose the prime numbers $(q_0, q_1, p) = (2^{34}+2^{27}+1, 2^{34}+2^{19}+1, 2^{38}+2^{23}+1)$ as the moduli. Each modulus has only three non-zero bits, such that a multiplication by them can be simplified as three shifts and additions.

B. Polynomial Processing Units (PPUs)

Beyond NTT and INTT, the majority of *CHAM* functions, including MULTPOLY, RESCALE, MULTMONO, AUTOMORPH, and MODSWITCH, are based on polynomial arithmetic. Hence, we design

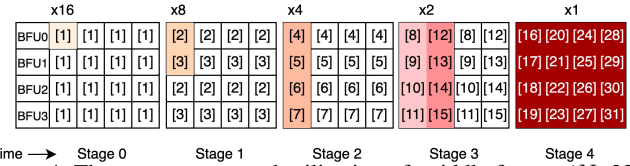


Figure 4: The arrangement and utilization of twiddle factors (N=32).

TABLE I: The polynomial arithmetic supported by *CHAM*.

Functions	Details
ModAdd(A, B)	$[a_0 + b_0, a_1 + b_1, ..., a_{N-1} + b_{N-1}]$
ModMul(A, B)	$[a_0 \times b_0, a_1 \times b_1, ..., a_{N-1} \times b_{N-1}]$
Rev(A)	$[a_{N-1}, ..., a_1, a_0]$
ShiftNeg(A, s)	$[a_{N-s}, ..., a_{N-1}, -a_1, ..., -a_{N-s-1}]$
Automorph(A, k)	$a_i \rightarrow (-1)^{\lfloor ik/N \rfloor} a_{ik \mod N}$

Note: $A = [a_0, a_1, ..., a_{N-1}]$ denotes the coefficients of a polynomial.

polynomial processing units (PPUs) for supporting these functions. In *CHAM*, PPUs also need to support RLWE-to-LWE and LWE-to-RLWE which involve conversion between polynomials and vectors. From the perspective of implementation, the coefficients of a polynomial are stored in a vector-like data structure, and all polynomial operations are carried out in a vectorized fashion. For this reason, both LWE ciphertext (composed of a vector and a scalar) and RLWE ciphertext (composed of polynomials) can be well supported by a unified data structure for both polynomials and vectors.

Table I lists the polynomial arithmetic implemented in *CHAM*. Except for coefficient-wise modular additions (ModAdd) and multiplications (ModMul), the function of Rev aims to reverse the order of the polynomial coefficients. ShiftNeg, serving as an underlying function for MultMono, RLWE-to-LWE, and LWE-to-RLWE, is implemented as a circular shift followed by a negation of the wrapped-around coefficients. Automorph is implemented as a permutation of the coefficients. It is noted that different from the automorphism unit proposed in *F1* [13], the Automorph task here executes in a serial fashion, while all PPUs are executed in parallel.

V. Evaluation

In this section, we implement *CHAM* on the Xilinx FPGA, and evaluate its performance based on a variety of benchmarks.

A. Implementation

For a fast prototyping, we implement *CHAM* on Xilinx U200 FPGA board and Intel Xeon W-2265 CPU@3.5 GHz. In particular, we use the Xilinx Vitis RTL flow for connecting *CHAM* and the host CPU. After verifying the functionality and stability of the prototyping system by running a large number of tests, we switch to Intel Xeon Gold 6130@2.1 GHz and Xilinx VU9P FPGA board for commercial production, with in-house FPGA platform and driver. *CHAM* is implemented at the frequency of 300 MHz with the floorplan shown in Fig. 5. Note that the initial floorplan utilizes too much BRAMs that imposes pressure on place and routing. To tackle this problem, we replace some BRAMs by URAM and LUTRAM to make the utilization rate of all of them below 75%. The final resource utilization is shown in Table II.

TABLE II: Resource utilization on the Xilinx VU9P FPGA.

Module	LUT	FF	BRAM	URAM	DSP
Compute Engine 0	259,318	89,894	640	294	986
Compute Engine 1	259,502	90,043	640	294	986
Platform	234,066	302,670	278	7	14
Total*	63.68%	20.41%	72.13%	61.98%	29.04%

* Measured by percentage in terms of the total FPGA resource.

B. Evaluation of Benchmarks

1) NTT: We evaluate the latency, resource utilization, and efficiency of NTT and INTT based on the Xilinx VU9P FPGA, with the results shown in Table III. The table shows three different implementation strategies for the twiddle-factor ROMs and the NTT local buffer, because the Xilinx platform allows a flexible use of either block RAMs (BRAMs) or LUT-based distributed RAMs (dRAMs).

Besides, we compare our implementation to existing work, *i.e.*, *HEAX* [31] and *F1* [13]. The NTT design of *HEAX* consumes the same clock cycles as ours. Nevertheless, our design is more compact due to the use of hardware-friendly moduli and constant-geometry dataflow. In addition, *CHAM* achieves high throughput because it has a total number of 60 NTT units which can perform 195 k operations per second (ops/sec), while *HEAX* performs 117 k ops/sec (assuming $N = 2^{12}$). A GPU-based work achieves an NTT throughput of 45 k ops/sec using a single CUDA kernel with 1024 threads, which however is much slower than our implementation. Finally, *F1*, implemented on ASICs, shows big performance advantage. However, if evaluated with FPGA platform, its NTT design would consume more than 65% DSP slices. Beyond the NTT, we also evaluate performance of key-switch operation. *CHAM* achieves a throughput of 65 k ops/sec that is 105× higher than the CPU baseline.

TABLE III: Comparison of a single NTT module.

Accelerator	Latency (l)	# Mult. (p)	ATP[1] ($l \times p$)	LUT[2] (u)	BRAM	ATP[1] ($l \times u$)
CHAM (BRAM only)	6144	4	1×	3324	14	1×
CHAM (BRAM+dRAM)[3]	6144	4	1×	6508	6	1.96×
CHAM (dRAM only)	6144	4	1×	9248	0	2.78×
HEAX [31]	6144	4	1×	22316	11	6.71×
F1 [13]	202	896	7.36×	-	-	-

[1] Area-time product is normalized. [2] *CHAM* deploys on Xilinx FPGAs with 6-input LUTs and 36 kbit BRAMs, while *HEAX* deploys on Intel FPGAs with 8-input LUTs and 20 kbit BRAMs. [3] Twiddle-factor ROM uses dRAM, local buffer uses BRAM.

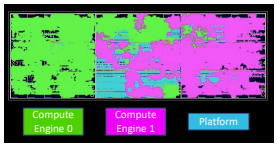


Figure 5: Floorplan result of *CHAM* based on Xilinx VU9P FPGA.

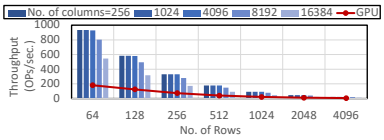
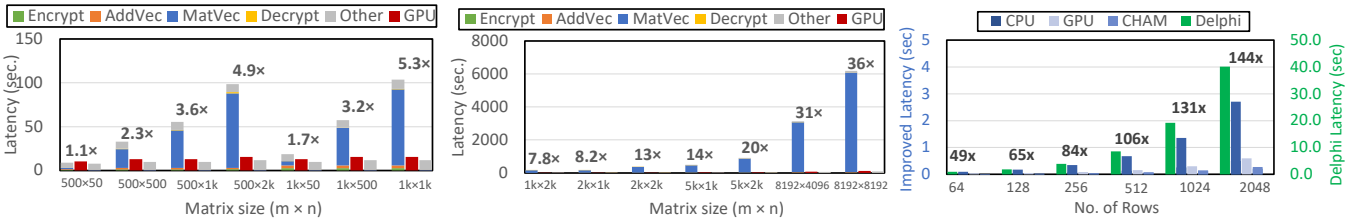


Figure 6: The throughput of *CHAM* for different matrices.

2) Matrix-vector product: We evaluate HMVP based on *CHAM* and measure the throughput as shown in Fig. 6. The throughput depends near-linearly on the number of rows (m) in the matrix. The number of columns (n) has less impact on the performance except for the cases of $n \geq m$ where throughput is degraded because a row, residing in multiple ciphertexts, needs to be aggregated. We compare *CHAM* to the CPU baseline with the results shown in Fig. 8. We observe that more than 90% computation has been offloaded to FPGA, resulting in >10× speed-up. We also observe that matrices with more rows demonstrate a higher performance gain. Next, we compare *CHAM* with the GPU implementation. In particular, *CHAM* demonstrates smaller latency ($0.3\times \sim 0.7\times$, as shown in Fig. 8) and higher throughput ($4.5\times$, as shown in Fig. 6) than the GPU.

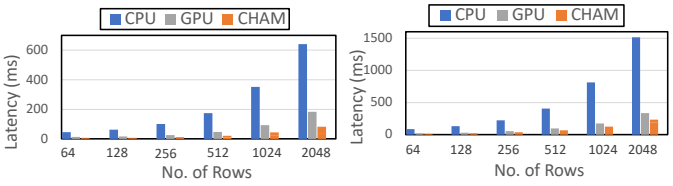
3) Logistic regression: We then evaluate performance for heterogeneous logistic regression (*HeteroLR*) where data is partitioned vertically across parties [16]. The *HeteroLR* trains a federated model based on overlapping samples provided by two parties $\mathcal{A}$ and $\mathcal{B}$. In the training process, party $\mathcal{A}$ and party $\mathcal{B}$ compute gradients based on their local data, while the arbiter then aggregates the gradients and distributes the updated gradient to them. Both $\mathcal{A}$ and $\mathcal{B}$ need to compute HMVP. The framework of *Federated AI Technology Enabler (FATE)* originally uses *Paillier*, a semi-HE algorithm. In this work, we replaced *Paillier* with B/FV for better utilizing the ability of hardware acceleration. Moreover, if combined with the techniques of mini-batch and matrix tiling, our algorithm is able to support data of any scale and be deployed in multiple hardware accelerators.

The evaluation of *HeteroLR* is conducted over datasets of different sizes, as shown in Fig. 7a and 7b. We conclude that B/FV reduces computation overhead of all steps in the *HeteroLR*, including encryption, vector addition (*add_vec*), matrix-vector product (*matvec*), and



(a) Logistic regression (b) Logistic regression (c) Beaver triple generation

Figure 7: Performance of LR/Beaver (Intel Xeon 6130@2.1 GHz vs. NVIDIA Tesla V100@1.29 GHz vs. Xilinx VU9P@300 MHz).



(a) No. of columns = 256 (b) No. of columns = 4096

Figure 8: Performance of HMVP (Intel Xeon 6130@2.1 GHz vs. NVIDIA Tesla V100@1.29 GHz vs. Xilinx VU9P@300 MHz).

decryption. Moreover, the HMVP, accelerated by *CHAM*, is faster than its CPU baseline by 30× to 1800×. Correspondingly, the end-to-end *HeteroLR* is accelerated by 2 to 36 times. The cases that involve large matrices (*e.g.*, 8192×4096 and 8192×8192) observe a high speed-up because for these cases matrix-vector product dominates the whole computation.

4) Beaver triple generation: In cryptographic neural-network inference, homomorphic encryption is used for generating multiplication triples (named *Beaver triples*) [28]. Since each matrix-vector multiplication of either party $\mathcal{A}$ or party $\mathcal{B}$ consumes a triple, a large number of triples need to be generated. This process can be significantly accelerated by *CHAM*. In particular, we improve the baseline algorithm of *Delphi* and evaluate it using *CHAM*. As shown in Fig. 7c, *CHAM* demonstrates a speed-up of 49× to 144× compared to the original implementation.

VI. Summary

In this work, we design an HE accelerator (*CHAM*) for high-performance homomorphic matrix-vector product. *CHAM* might be the first HE accelerator deployed for commercial applications of federated learning and multi-party computation. Different from existing HE accelerators, *CHAM* employs an approach of algorithm-hardware co-design. The experimental results demonstrate 1800× speed-up for matrix-vector product, 36× speed-up for logistic regression, and 144× speed-up for Beaver triple generation.

References

[1] A. Al Badawi *et al.*, "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, 2021.

[2] P. G. M. Alves *et al.*, "Faster Homomorphic Encryption over GPGPUs via Hierarchical DGT," in *IACR Cryptology ePrint Archive*, 2021.

[3] F. Boemer *et al.*, "nGraph-HE2: A High-throughput Framework for Neural Network Inference on Encrypted Data," in *ACM Conference on Computer and Communications Security*, 2019.

[4] C. Boura *et al.*, "CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes," *Journal of Mathematical Cryptology*, vol. 14, no. 1, 2020.

[5] A. Brutzkus *et al.*, "Low Latency Privacy Preserving Inference," in *ICML*, 2019.

[6] D. D. Chen *et al.*, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Transactions on Circuits and Systems I*, vol. 62-I, no. 1, 2015.

[7] H. Chen *et al.*, "Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts," in *IACR Cryptology ePrint Archive*, 2021.

[8] J. H. Cheon *et al.*, "Homomorphic Encryption for Arithmetic of Approximate Numbers," *IACR Cryptology ePrint Archive*, 2017.

[9] I. Chillotti *et al.*, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *Journal of Cryptology*, vol. 33, no. 1, 2020.

[10] W. Dai *et al.*, "cuHE: A Homomorphic Encryption Accelerator Library," in *IACR Cryptology ePrint Archive*, 2016.

[11] N. Dowlin *et al.*, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *ICML*, 2016.

[12] J. Fan *et al.*, "Somewhat Practical Fully Homomorphic Encryption," in *15th International Conference on Practice and Theory in Public Key Cryptography*, 2012.

[13] A. Feldmann *et al.*, "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption," in *Annual International Symposium on Microarchitecture*, 2021.

[14] R. Geelen *et al.*, "BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption," in *ACM Conference*, 2022.

[15] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, 2009.

[16] S. Hardy *et al.*, "Private Federated Learning on Vertically Partitioned Data via Entity Resolution and Additively Homomorphic Encryption," 2017. [Online]. Available: http://arxiv.org/abs/1711.10677

[17] E. Hesamifard *et al.*, "CryptoDL: Deep Neural Networks over Encrypted Data," 2017. [Online]. Available: http://arxiv.org/abs/1711.05189

[18] Z. Huang *et al.*, "Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference," *USENIX Security*, 2022.

[19] W. Jung *et al.*, "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs," *IACR Transactions on CHES*, vol. 2021, no. 4, 2021.

[20] W. Jung *et al.*, "Accelerating Fully Homomorphic Encryption through Architecture-centric Analysis and Optimization," *IEEE Access*, 2021.

[21] C. Juvekar *et al.*, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *27th USENIX Security Symposium*, 2018.

[22] J. Kim *et al.*, "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," 2022. [Online]. Available: http://arxiv.org/abs/2205.00922

[23] S. Kim *et al.*, "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption," in *ISCA*, 2022.

[24] S. Kim *et al.*, "FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption," in *International Conference on Reconfigurable Computing and FPGAs, ReConFig*, 2019.

[25] S. Kim *et al.*, "Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme," in *IEEE International Symposium on FCCM*, 2020.

[26] W. Lu *et al.*, "PEGASUS: Bridging Polynomial and Non-Polynomial Evaluations in Homomorphic Encryption," in *IEEE Symposium on Security and Privacy*, 2021.

[27] V. Migliore *et al.*, "Hardware/Software Co-Design of an Accelerator for FV Homomorphic Encryption Scheme Using Karatsuba Algorithm," *IEEE Transactions on Computers*, vol. 67, no. 3, 2018.

[28] P. Mishra *et al.*, "DELPHI: A Cryptographic Inference Service for Neural Networks," in *29th USENIX Security Symposium*, 2020.

[29] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *Journal of ACM*, vol. 15, no. 2, 1968.

[30] B. Reagen *et al.*, "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference," in *HPCA*, 2021.

[31] M. Sadegh Riazi *et al.*, "HEAX: An Architecture for Computing on Encrypted Data," in *ASPLOS*, 2020.

[32] N. Samardzic *et al.*, "CraterLake : A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data," in *ISCA*, 2022.

[33] S. Sinha Roy *et al.*, "FPGA-Based High-performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in *HPCA*, 2019.