2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 979-8-3503-4963-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ASAP61560.2024.00023

A Framework for Generating Accelerators for Homomorphic Encryption Operations on FPGAs

Yang Yang[1], Rajgopal Kannan[2] and Viktor K. Prasanna[1]
[1]Department of Electrical and Computer Engineering, University of Southern California
[2]DEVCOM Army Research Office
Email: {yyang172, prasanna}@usc.edu, rajgopal.kannan.civ@army.mil

***Abstract*—Homomorphic Encryption (HE) is a promising technique for preserving user data privacy in cloud computing. Nevertheless, HE operations are magnitudes slower than unencrypted computations due to their high computational complexity. FPGAs are attractive platforms for designing domain-specific accelerators. However, manually programming FPGAs for HE applications is non-trivial because of the vastly different parameter settings and latency requirements. To close the gap, we propose a framework to generate low latency FPGA accelerators for all the operations supported by HE, enabling users to utilize FPGA-accelerated HE processing without requiring knowledge of FPGA implementation details. The framework takes HE parameters and hardware resource constraints as input, uses design space exploration to automatically determine the design parameters that minimize HE computation latency, and produce synthesizable Verilog code. We propose a layered approach that decomposes HE operations into basic HE primitives, coupled with a parameterized HE domain-specific architecture that can efficiently execute the HE primitives. This approach avoids allocating dedicated FPGA resources to different subroutines within HE operations and improves compute utilization. Our evaluation shows that the generated accelerators significantly reduce latency in various HE operations, achieving up to $215\times$ improvement over state-of-the-art CPU implementations. We demonstrate our framework's capability to compose end-to-end HE applications using HE CNN inference. Our designs outperform state-of-the-art CPU designs in latency by up to $60\times$.**

***Index Terms*—homomorphic encryption, FPGA acceleration framework, design space exploration**

I. Introduction

Utilizing Homomorphic Encryption (HE) to perform computations directly on encrypted data has gained significant attention in recent years [1], [2], [3], [4], [5]. However, computation using HE is orders of magnitude slower than computation on unencrypted data. All HE computations take place within a polynomial ring with a large modulus, typically reaching thousands of bits [6]. Consequently, HE computations have high computational complexity and require extensive memory usage [7]. These challenges necessitate the need for domain-specific accelerators tailored for HE computations [8].

Field Programmable Gate Arrays (FPGAs) [9], [10] are being adopted rapidly in public clouds to improve performance and reduce application deployment cost [11], [12]. With the fine grained programmable architecture of FPGAs, they are well suited for developing accelerators tailored for dedicated applications. Data intensive applications such as HE require high memory bandwidth to supply data to the processing elements. High-Bandwidth Memory (HBM) has been deployed by FPGA vendors to address the need for faster data movement [9], [10]. The latest HBM-enabled FPGAs can provide up to 460 GB/s bandwidth and 16 GB capacity.

However, it is non-trivial to efficiently utilize the resources offered by state-of-the-art FPGAs. Firstly, the design space for HE hardware acceleration is large [13]. HE applications [1], [3], [4] use a wide range of HE parameters (Section II). It requires significant development effort to comprehensively support a diverse set of HE parameters under varying FPGA resource constraints. Secondly, effectively mapping HE operations onto FPGAs to maximize compute utilization is not an easy task. Previous FPGA designs and frameworks for HE allocate dedicated computation resources for different subroutines within an HE operation [14], [15]. Their approaches lead to underutilized resources and increased latency, as resources allocated to one subroutine cannot be utilized for others and remain idle when not in use.

Motivated by the challenges, we propose a framework to automatically generate FPGA accelerators for low latency execution of HE operations. The framework takes HE parameters and resource constraints as input. It performs design space exploration to identify design options that minimize HE computation latency and generates Verilog code as the output. Our framework utilizes a layered approach combined with a parameterized HE domain-specific architecture to construct FPGA implementations, enabling effective reuse of compute datapaths. The framework decomposes HE operations into HE subroutines, which are composable components in our framework and determine the processing dataflow. The HE subroutines are divided into HE primitives, including modular arithmetic logic and permutation units, which form the basic building blocks for HE computation. The contributions of this paper are:

- We present a framework for FPGA acceleration of HE operations. Our framework significantly boosts productivity by automatically generating FPGA designs for various HE parameters and resource constraints.
- The framework utilizes a layered approach to improve compute utilization during the execution of HE operations. We implement dataflow specifically optimized for various HE subroutines. We further minimize memory data transfers by fusing the processing of HE subroutines.
- We develop a design space exploration (DSE) tool for

2160-052X/24/$31.00 ©2024 IEEE
DOI 10.1109/ASAP61560.2024.00023

effective FPGA resource allocation and latency estimation. The DSE is used to identify the design parameters without the need to run through the time-consuming FPGA implementation flow.
- Our framework generates FPGA accelerators for various HE operations, achieving up to $215\times$ latency reduction compared to State-Of-The-Art (SOTA) CPU implementations. We use HE CNN inference to demonstrate our framework's capability for composing end-to-end HE applications. Our designs outperform SOTA CPU designs in latency by up to $60\times$.

II. Background

A. Homomorphic Encryption (HE)

Our framework targets the CKKS scheme [16]. The proposed techniques can be applied to other HE schemes [17], [18]. The CKKS parameters are listed in Table I. A cleartext vector m of $N/2$ complex numbers is first encoded as a plaintext polynomial (*pt*). This *pt* is then encrypted into a pair of ciphertext polynomials (*ct*), $[\![m]\!] = (a, b)$, using a secret key s. Both *pt* and *ct* have a polynomial degree of $N-1$. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$. The field of a fresh ciphertext is $\mathcal{R}_Q^2 = (\mathcal{R}/Q\mathcal{R})^2$, which means a pair of polynomials with coefficients from $\mathbb{Z}_Q$, i.e., integers modulo Q. The modulus Q is chosen to be sufficiently large (hundreds or even thousands of bits) to avoid data corruption during encrypted computation. The Residue Number System (RNS) is used to manage this complexity [16]. Specifically, let $Q = \prod_{i=0}^{L} q_i$ be the modulus of a ciphertext with a computational level L. A polynomial in $\mathcal{R}_Q$ has $L+1$ limbs, where the coefficients of the i-th limb are in $\mathbb{Z}_{q_i}$. HE operations increase the encryption noise and reduce the computational level (L) of ciphertext. When the level reaches zero, decryption becomes impossible. Bootstrapping can reset the noise and enable Fully Homomorphic Encryption (FHE). As the number of HE operations is a known priori in many real-world HE applications, bootstrapping can be avoided in accordance with prior works [19], [20], [21], [22], [23]. The security level (λ) of the CKKS scheme relies on polynomial degree N and ciphertext moduli Q.

TABLE I
CKKS Parameters

Parameter	Description
N	Degree of the plaintext and ciphertext polynomials.
L	Maximum computational level of a ciphertext.
l	Current level of a ciphertext, $0 \le l \le L$. A ciphertext with level l has $l+1$ limbs.
Q	Maximum modulus of a ciphertext coefficient. It is represented by $L+1$ co-primes q_i. $Q = \prod_{i=0}^{L} q_i$.
P	Product of k additional co-primes p_i for the raised modulus. $P = \prod_{i=0}^{k-1} p_i$
dnum	Number of digits in the switching key.
α	Number of limbs of a single digit during key switching decomposition. $\alpha = \lceil (L+1)/\mathsf{dnum} \rceil$.
β	An l-limb ciphertext is split into β digits during key switching decomposition, where $\beta = \lceil (l+1)/\alpha \rceil$.

B. Homomorphically Encrypted (HE) Operations

CKKS supports the following operations.

- PtCtAdd and PtCtSub perform element-wise addition and subtraction between a plaintext m_0 and a ciphertext $[\![m_1]\!]$ respectively.
- CtCtAdd adds two encrypted vectors $[\![m_0]\!]$, $[\![m_1]\!]$ and outputs the encryption of the element-wise sum of the two vectors, $[\![m_0+m_1]\!]$. CtCtSub performs element-wise subtraction and produces $[\![m_0-m_1]\!]$.
- PtCtMult multiplies a plaintext m_0 with a ciphertext $[\![m_1]\!]$ and outputs $[\![m_0 \cdot m_1]\!]$, where $\cdot$ denotes element-wise multiplication of the two vectors.
- CtCtMult performs element-wise multiplication between two ciphertext $[\![m_0]\!]$ and $[\![m_1]\!]$ and outputs $[\![m_0 \cdot m_1]\!]$.
- Let $[\![m]\!]$ be a ciphertext of vector $(c_0, c_1, c_2, ..., c_n)$, where $n = N/2$. Rotate by r slots outputs $[\![\phi_r(m)]\!] = (c_r, c_{r+1}, ..., c_{n-1}, c_0, c_1, ..., c_{r-1})$, i.e., the elements of vector are circularly shifted by r slots. Rotate is not a simple coefficient shift of ciphertext polynomials.
- The Rescale operation adjusts the scaling factors of the ciphertext, to prevent noise overflow [16].

CtCtMult is carried out by performing multiplications between limbs of input ciphertext (LimbMult). The output of LimbMult is encrypted with secret key s^2. KeySwitch is used to homomorphically change the encryption key back to s. The complete algorithm is outlined in Algorithm 1. The first subroutine of KeySwitch is Decomp (Line 4), where the l limbs are divided into β digits (groups), each containing α limbs. Subsequently, through the ModUp subroutine, each group is expanded to $\alpha\beta+k-1$ limbs. The key switching key ($\mathsf{ksk}_{s^2\to s}$) is comprised of a matrix of polynomials with dimensions $2\times\beta$, where each matrix element contains $\alpha\beta+k-1$ limbs. The InnerProd subroutine performs matrix-vector multiplication of limbs between an input vector of size β limbs and the $\mathsf{ksk}_{s^2\to s}$. This is followed by the ModDown subroutine, which reduces the limb count of the InnerProd output from $\alpha\beta+k-1$ to $l+1$ limbs. Finally, a limb modular addition is performed to produce the output ciphertext (LimbAdd, Line 8). Decomp, ModUp, InnerProd and ModDown are described in Algorithm 3, 2, 10, and 4 respectively in [24]. Ciphertext limbs are in evaluation domain by applying the Number Theoretic Transform (NTT). BaseConv operation within ModUp and ModDown requires that the ciphertext representation be in polynomial form [16]. Therefore INTT and NTT are used at the beginning and the end of ModUp and ModDown [24].

Algorithm 1 CtCtMult($[\![\mathsf{m_0}]\!], [\![\mathsf{m_1}]\!], \mathsf{ksk}_{s^2\to s}$)

1: $(a_0, b_0) \leftarrow [\![m_0]\!]$
2: $(a_1, b_1) \leftarrow [\![m_0]\!]$
3: $(a_2, b_2, c_2) \leftarrow (a_0a_1, a_0b_1 + a_1b_0, b_0b_1)$ ▷ LimbMult
4: $a_2 \leftarrow \mathsf{Decomp}_\beta(a_2)$
5: $\hat{a_2} \leftarrow \mathsf{ModUp}(a_2)$
6: $x, y \leftarrow \mathsf{InnerProd}(\mathsf{ksk}_{s^2\to s}, \hat{a_2})$
7: $\hat{x}, \hat{y} \leftarrow \mathsf{ModDown}(x), \mathsf{ModDown}(y)$
8: **return** $(b_2 + \hat{x}, c_2 + \hat{y})$ ▷ LimbAdd

The first step of Rotate is the Automorph subroutine, which is performed on each limb coefficients via a permutation mapping $i \mapsto \sigma_r(i)$, where $\sigma_r(i) = i \cdot 5^r \mod N, 0 \le i \le N-1$,

i.e., limb coefficient c_i becomes coefficient $c_{\sigma_r(i)}$. The output ciphertext of Automorph is encrypted in a different secret key, requiring the same KeySwitch subroutine as in CtCtMult to obtain a valid output. The Rescale operation can be considered as a special implementation of ModDown subroutine in which the number of limbs is reduced by one (i.e., from l to $l-1$). For a more formal description of various subroutines, we refer readers to [16].

III. Framework Design Methodology

A. Decomposition of HE Operations

Developing an FPGA framework for a broad range of HE parameters and resource constraints is a complex task. Prior approaches of manually adjusting each FPGA design for specific HE operations are time-consuming and inefficient for low latency execution [25], [26]. Having dedicated hardware blocks for different HE subroutines [14] is also unsuitable for low-latency execution because HE subroutines cannot always run in parallel [24], leading to hardware underutilization and long latency. We propose a layered approach to support all HE operations in a scalable manner. Our approach enables the efficient reuse of computation datapaths across different subroutines within an HE operation. Our approach defines three layers to decompose an HE operation into their fundamental components. The top layer, defined as *API Layer*, includes HE operations that are exposed to the users of our framework. The HE operations are then decomposed into HE subroutines in the *Intermediate Layer*. Finally, each HE subroutine is broken down into HE primitives in the *Foundational Layer*.

1) API Layer – HE Operations: The API Layer comprises of application level HE operations and their parameters (Section II-B). This layer selects the appropriate HE subroutines for each operation and implements a top-level control module. The control module executes control sequences that run the HE subroutines according to the algorithm of each HE operation. It also enables the fusion of HE subroutines if the on-chip SRAMs is large enough to store the intermediate data between HE subroutines.

2) Intermediate Layer – HE Subroutines: The Intermediate Layer comprises a variety of *HE subroutines*, such as Ntt, Intt, Automorph, ModUp, ModDown, Decomp, BaseConv, and KskInnerProd. An HE subroutine is defined as an operation in which the datapath is implemented through multiple HE primitives while the operation is not exposed directly to the users of the framework. The main objective of this layer is to generate control sequences for each HE subroutine. The control sequences manage the datapaths and oversee the data movement between external memory and on-chip SRAMs.

3) Foundational Layer – HE Primitives: In the Foundational Layer of our framework, we identify two basic *HE primitives*: ModAlu and Permute. The primitives serve as the building blocks for constructing datapaths of the components in the higher level layers. ModAlu is responsible for executing modular arithmetic, and Permute performs arbitrary permutation of vectors.

B. High Level Design of the Framework

Figure 1 shows the workflow of our framework. It takes HE parameters and resource constraints as input and produces Verilog code of an FPGA accelerator that can execute the specified HE operations. HE parameters are specified by:

- CKKS Parameters ($\mathcal{P}$): The CKKS parameters (Table I) determine the specific algorithm to be accelerated.
- HE Operations ($\mathcal{OP}$): The HE computations are represented by a dependency graph. Each node in the graph corresponds to one of the operations defined in the API Layer.

Hardware resource constraints are specified by:

- DSP ($\mathcal{D}_{max}$) and SRAM ($\mathcal{R}_{max}$) constraints: DSPs and on-chip SRAMs that can be used by the generated hardware. $\mathcal{R}_{max}$ is specified by the total available URAM ($\mathcal{R}_{uram\_max}$) and BRAM ($\mathcal{R}_{bram\_max}$) instances.
- I/O bandwidth ($\mathcal{BW}_{max}$): Available read and write bandwidth of the target FPGA design. The bandwidth is used to transfer input, output and intermediate data on the *Memory Interface* of the generated hardware.
- Metadata ($\mathcal{M}$): Platform related metadata on the target FPGA device, such as the bit width of DSP and memory organization of the on-chip SRAMs. This is used by the design space exploration to estimate the resource consumption of different hardware modules.

Design Space Explorer (DSE) (Section V) identifies design parameters that minimize the latency for executing $\mathcal{OP}$. The RTL Design Generator takes the design parameters and applies the proposed layered approach to generate Verilog code. It first generates the top-level control module, which establishes the dataflow across HE subroutines. Subsequently, control sequences and dataflows for each individual HE subroutine are produced, followed by the instantiation of the computation datapaths. The generated FPGA accelerator processes HE operations sequentially, one node at a time.

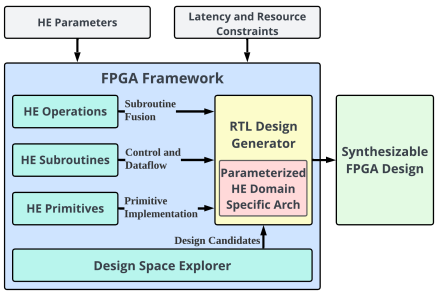

Fig. 1. High level workflow of the proposed framework.

1) Design Parameters: Our framework utilizes an HE domain specific architecture (Section IV) to execute all HE operations. The architecture is parameterized to support diverse resource constraints. The RTL Design Generator interacts with the DSE using a set of *design parameters*, each of which can be adjusted to tune the computation latency while also impacting FPGA resource usage. The design parameters are:

- Number of Modular ALUs (*NumAlu*): Number of modular arithmetic logic units in the design. The ModALUs are

fully pipelined. Each can perform one modular addition, subtraction, and multiplication per cycle.
- Throughput of Permute Pipeline (*PermTput*): This parameter determines the throughput for each permute pipeline. The permute pipeline can accept and produce *PermTput* elements per cycle in steady state.
- Capacity of Scratchpad (*ScratchSize*): Total size of the scratchpad memory implemented using FPGA URAMs and BRAMs [27], [10].

IV. Hardware Design and Dataflow

A. Architecture Overview

Our framework utilizes a parameterized HE domain specific architecture to support diverse HE parameters and resource constraints. The high level architecture is depicted in Figure 2. It consists of an array of modular ALUs and a permute pipeline, corresponding to the components in the HE primitives. A scratchpad memory is implemented to store intermediate limbs and twiddle factors for Ntt and Intt. The scratchpad memory has multiple banks to enable parallel accesses from multiple requesters. The scratchpad is connected to a Memory Interface module that oversees data transfer between the scratchpad and external memory. The architecture includes two control units: the HE Subroutines Control module, which orchestrates control sequences for processing each HE subroutine, and the HE Operations Control module, which manages the dataflows across subroutines.

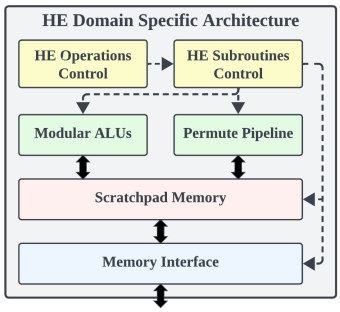

Fig. 2. Architecture of the FPGA design generated by the framework.

B. Hardware Implementation Details

1) Modular ALUs: The modular ALUs support modular additions, subtractions, and multiplications. The bit width for input and output operands is defined by $\max_{0<i<L} \log q_i$. For modular multiplication, we implement the Barrett reduction algorithm [28]. Our modular multiplier implementation is based on a prior fully pipelined design [29]. The modular multiplier implementation includes three integer multipliers: the first two perform full-width multiplications, combining two input operands to produce the complete output; the remaining one is a half-width multiplication, producing the lower half of the output. The modular ALUs have a throughput of processing *NumAlu* coefficients per cycle.

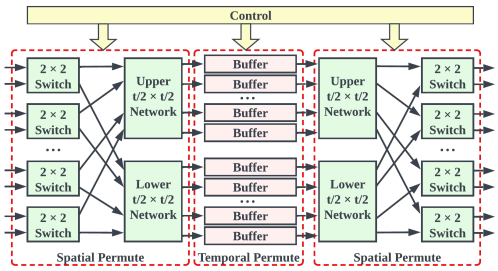

Fig. 3. Architecture of the permute pipeline.

2) Permute Pipeline: Ntt, Intt and Automorph require permutation of coefficients for individual limbs. The permute pipeline enables arbitrary permutations of limb coefficients. Due to the complex data access patterns, these operations are difficult to process in parallel [15]. Previous solutions use fully connected crossbars with carefully designed data placement to prevent scratchpad bank conflicts [26], [30]. We adopt the Streaming Permutation Network (SPN) [31] for parallel permutation of coefficients without the need for an expensive crossbar. The architecture of an SPN is illustrated in Figure 3. It consists of three subnetworks – two spatial permutation networks and one temporal permutation network. The spatial permutation shuffles $t = PermTput$ coefficients received in the same cycle, while the temporal permutation rearranges coefficients across N/t cycles. Each spatial permutation network has $\log t$ stages, using 2×2 switches recursively to create a t-to-t connection. Temporal permutation is achieved by read and write operations to t buffers with pre-computed addresses. Each buffer stores N/t coefficients.

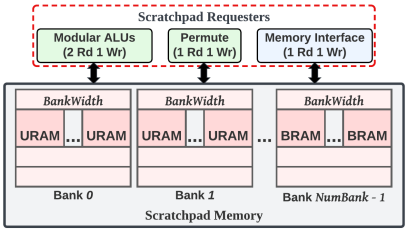

Fig. 4. Architecture of the scratchpad memory and its requesters.

3) Scratchpad Memory: Figure 4 shows the architecture of the scratchpad. It has multiple banks to store intermediate results and Ntt/Intt twiddle factors. This memory is configured as $ScratchSize = NumBank \times BankWidth \times BankDepth$. The value of *BankWidth* is set to match the peak throughput of the computation datapaths, ensuring that one bank can supply $\max(NumAlu, PermTput)$ coefficients per cycle. *NumBank* is set based on the requirements of its requesters, ensuring that all requesters can concurrently access the scratchpad. The modular ALUs require two read ports and one write port. The permute pipeline and memory interface need one read port and one write port each. *ScratchSize* is adjusted by the DSE by varying *BankDepth*.

C. Dataflow Design

HE subroutines have two distinct computation patterns: the *Limb Pattern* and the *Coefficient Pattern* [32]. In the Limb

Pattern, the i-th output limb is derived from the corresponding limb in the input ciphertext. In contrast, the Coefficient Pattern uses the i-th coefficient from all limbs to compute the i-th coefficient of an output limb. Figure 5 illustrates the two computation patterns. In the diagram, a ciphertext's row represents a limb, and each square box within represents a coefficient. For computing the output coefficient highlighted in red, the Limb Pattern operates on a specific "row" of the input, while the Coefficient Pattern processes a "column" of coefficients. HE subroutines following the Limb Pattern include Automorph, Ntt, Intt, Decomp, and KskInnerProd, while ModUp and ModDown falls under the Coefficient Pattern. We design two dataflows tailored to the computation patterns.

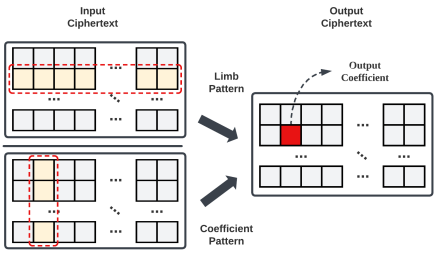

Fig. 5. Two computation patterns of HE subroutines.

1) Limb Dataflow: In Limb Dataflow, the modular ALUs calculate one output limb at a time, with each ALU handling $N/NumAlu$ coefficients. The permute pipeline also handles one limb at a time for permutation. Data movement mirrors this limb-focused approach, sequentially transferring coefficients from a single limb.

2) Coefficient Dataflow: The coefficient dataflow operates in a different order. It starts by fetching the first set of $2 \cdot L \cdot NumAlu$ coefficients from all input limbs, performs the necessary computations, and then stores $2 \cdot L \cdot NumAlu$ output coefficients. This repeats for each subsequent $N/NumAlu$ set of coefficients.

3) HE Subroutine Fusion: To reduce data transfer on the memory interface, the top-level dataflow enables fusion of consecutive HE subroutines using Limb Dataflow. The output limb from one HE subroutine is immediately used as the input for the next, eliminating the need to write intermediate outputs back to external memory and then re-fetch them. However, when transitioning to Coefficient Dataflow, intermediate data is saved to external memory (via the memory interface) before the execution of the next HE subroutine.

V. Design Space Exploration

Design Space Exploration (DSE) generates design candidates based on the input to the framework. Since both SPN and the scratchpad require BRAM/URAM resources, DSE allocates these resources by tuning the *PermTput* and *ScratchSize* parameters. Additionally, simply maximizing the usage of DSP resources is not necessary if other system components become bottlenecks. To reduce DSE complexity, we use the parametric search through doubling (approximate) algorithm as defined in Algorithm 2. We iterate the design parameters in ascending order, ensuring that designs with lower resource requirements are selected when multiple designs have the same latency. DSE iterates through possible values of *NumAlu* and *PermTput* in power-of-two steps to reduce design complexity. As discussed in Section IV-B3, the scratchpad bank width is determined based on *NumAlu* and *PermTput*. DSE then determines the *ScratchSize* increment step by considering the BRAM/URAM properties, number of banks and width of each bank (Line 4). Latency and resource estimators are invoked to project the latency and resource consumption of each design candidate (Line 6). DSE stores the design parameters if the latency is currently minimal and resource consumption satisfies the constraints (Line 7–9).

Algorithm 2 Design Space Exploration

Input: $\mathcal{P}$, $\mathcal{OP}$, $\mathcal{D}_{max}$, $\mathcal{R}_{max}$, $\mathcal{BW}_{max}$, $\mathcal{M}$
Output: *NumAlu*, *PermTput*, *ScratchSize*

1: $\mathcal{T}_{min} \leftarrow \infty, i \leftarrow 0, j \leftarrow 1$
2: **while** $NumAlu \leftarrow 2^i$ **do** ▷ Continue until exceeding 512
3: **while** $PermTput \leftarrow 2^j$ **do** ▷ Continue until exceeding 512
4: $ScratchIncStep \leftarrow f(NumAlu, PermTput, \mathcal{M})$
5: **for** $ScratchSize \leftarrow (0, \mathcal{R}_{max}, ScratchIncStep)$ **do**
6: Compute $\mathcal{T}, \mathcal{D}, \mathcal{R}$
7: **if** $\mathcal{T} < \mathcal{T}_{min}, \mathcal{D} \leq \mathcal{D}_{max}, \mathcal{R} \leq \mathcal{R}_{max}$ **then**
8: $\mathcal{T}_{min} \leftarrow \mathcal{T}$
9: Store *NumAlu*, *PermTput*, *ScratchSize*
10: $j = j + 1$
11: $i = i + 1$

A. Latency Estimator

Let $\mathcal{O}$ be the set of HE operations in $\mathcal{OP}$. The total latency $\mathcal{T}$ of $\mathcal{OP}$ is sum of the execution times of all the operations in $\mathcal{O}$. Let $\mathcal{S}_i$ be the set of HE subroutines in HE operation i. $\mathcal{T}$ is defined by Equation 1, where $\mathcal{T}_{\mathcal{S}_{ij}}$ is the latency of subroutine j in HE operation i.

$$\mathcal{T} = \sum_{i \in \mathcal{O}} \sum_{j \in \mathcal{S}_i} \mathcal{T}_{\mathcal{S}_{ij}} \tag{1}$$

The latency for each $\mathcal{T}_{\mathcal{S}_{ij}}$ is determined based on Equation 2, where $\mathcal{T}_{\mathcal{S}_{ij}\_compute}$ and $\mathcal{T}_{\mathcal{S}_{ij}\_memory}$ refer to the compute and memory time respectively.

$$\mathcal{T}_{\mathcal{S}_{ij}} = \max(\mathcal{T}_{\mathcal{S}_{ij}\_compute}, \mathcal{T}_{\mathcal{S}_{ij}\_memory}) \tag{2}$$

Compute Time Estimation: $\mathcal{T}_{\mathcal{S}_{ij}\_compute}$ is estimated by dividing the total required computations by the throughput of the execution pipeline. Due to space limitation, we describe the compute time estimation for Ntt/Intt (Limb Pattern) and BaseConv (Coefficient Pattern). We omit the details for other HE subroutine as they follow a similar methodology. Each stage of Ntt and Intt, there are $N/2$ multiplications, additions, and subtractions, with a total of $\log N$ stages. Thus,

$$\mathcal{T}_{limb\_\text{Ntt/Intt}} = \log N \cdot \max\left(\frac{3 \cdot N}{2 \cdot NumAlu}, \frac{N}{PermTput}\right) \tag{3}$$

The total compute time for Ntt/Intt is then obtained by $\mathcal{T}_{\text{Ntt/Intt}} = \mathcal{T}_{limb\_\text{Ntt/Intt}} \cdot NumLimbs$, where *NumLimbs* is the number of limbs processed. Assuming that a BaseConv is required to process l' input limbs and reduces them to l output

limbs, the computation for each output coefficient involves $(2l'+2)$ modular multiplications and one modular addition. Therefore the total computation time for BaseConv is:

$$\mathcal{T}_{\mathsf{BaseConv}} = \frac{l \cdot N \cdot (2l'+3)}{\mathit{NumAlu}} \tag{4}$$

Memory Time Estimation: We calculate the memory time for an HE subroutine as per Equation 5.

$$\mathcal{T}_{\mathcal{S}_{ij}\_\mathit{memory}} = \frac{\mathcal{B}_{\mathit{read}} + \mathcal{B}_{\mathit{write}}}{\mathcal{BW}_{\mathit{max}}} \tag{5}$$

$\mathcal{B}_{\mathit{read}}$ and $\mathcal{B}_{\mathit{write}}$ represent the total memory read and write bytes. They are calculated by summing the data transfers required for the subroutine, including transferring limbs, twiddle factors and key switching keys. Limb read and write bytes are estimated based on the algorithm of specific HE subroutine.

Our framework reduces memory transfers through HE subroutine fusion, which is applicable to CtCtMult, Rotate, and KeySwitch. For instance, Automorph can be fused with Decomp if sufficient space exists in the scratchpad, avoiding the memory transfers for the intermediate limbs. The DSE assesses if the scratchpad can store the output limbs of the preceding subroutine. If so, it eliminates the corresponding limb bytes from the $\mathcal{B}_{read}$ or $\mathcal{B}_{write}$ calculations.

B. Resource Estimator

For a given set of design parameters, we calculate the DSP ($\mathcal{D}$), URAM ($\mathcal{R}_{\mathit{uram}}$) and BRAM ($\mathcal{R}_{\mathit{bram}}$) usage for the generated design. The estimated resource consumption is then compared against available constraints to ensure it does not exceed the limit.

DSP resources ($\mathcal{D}$): DSP usage is defined as $\mathcal{D} = \mathit{NumAlu} \cdot d_{ALU}$. It depends on the implementation of the modular ALUs. The factor d_{ALU} refers to the number of DSPs required for a single modular ALU. Only the modular multiplier utilizes DSPs for its implementation.

URAM and BRAM resources ($\mathcal{R}$): We use BRAM to construct the buffers for the SPN. Each permute pipeline has *PermTput* buffers. Each entry of the buffer stores one coefficient and each buffer is $N/\mathit{PermTput}$ deep. Each SPN buffer requires multiple BRAMs to be chained horizontally and/or vertically. Horizontal chaining occurs when bits per coefficient (b_{coef}) exceeds the bits per row of a BRAM (b_{bram}). Vertical chaining is used when $N/\mathit{PermTput}$ is greater than the number of rows per BRAM (r_{bram}). The BRAM consumption for the permute pipeline ($\mathcal{R}_{\mathit{spn\_bram}}$) is

$$\mathcal{R}_{\mathit{spn\_bram}} = \lceil \frac{b_{\mathit{coef}}}{b_{\mathit{bram}}} \rceil \cdot \lceil \frac{N}{\mathit{PermTput} \cdot r_{\mathit{bram}}} \rceil \cdot \mathit{PermTput} \tag{6}$$

In this equation, the first term determines the BRAM instances for each buffer to supply b_{coef} bits per cycle. The second term calculates the BRAM instances given the depth of each buffer.

The scratchpad memory is implemented using URAMs and BRAMs. We calculate the number of URAM instances required for the scratchpad using Equation 7, where b_{uram} refers to the bits per URAM row and r_{uram} is the depth per URAM instance.

$$\mathcal{R}_{\mathit{uram}} = \mathit{NumBank} \cdot \lceil \frac{\mathit{BankWidth}}{b_{\mathit{uram}}} \rceil \cdot \lceil \frac{\mathit{BankDepth}}{r_{\mathit{uram}}} \rceil \tag{7}$$

BRAM estimation, denoted as $\mathcal{R}_{\mathit{scratch\_bram}}$, follows a similar approach. The total BRAM instances required is $\mathcal{R}_{\mathit{bram}} = \mathcal{R}_{\mathit{spn\_bram}} + \mathcal{R}_{\mathit{scratch\_bram}}$.

VI. Experiments

A. Experimental Setup

Platform: We evaluate our framework on an AMD Alveo U280 FPGA. The FPGA has 1,304K LUTs, 2,607K FFs, 41 MB on-chip SRAM and 9,024 DSPs. We perform synthesis, place-and-route using Vivado 2021.1. The results are reported after place-and-route. Each of the designs generated through our framework is configured to operate at a frequency of 250 MHz. In order to evaluate the efficiency of our framework across a variety of hardware platforms, we conduct experiments under different resource constraints.

TABLE II
HE Parameters for the evaluation of HE operations.

Parameter Set	N	L	dnum	k	λ
Set-1	2^{16}	44	45	1	98
Set-2	2^{16}	23	3	8	128
Set-3	2^{14}	8	1	1	128

HE Parameters: We use three sets of HE parameters to evaluate HE operations as listed in Table II. These parameters are commonly used in real-world HE applications [33], [24], [26]. Set-1 and Set-2 are large HE parameters that provides more computational levels while Set-3 is a small set of HE parameters. All sets of parameters meet $\lambda \geq 96$ bits security level and use 32-bit RNS prime integers [34], [35].

B. Framework Evaluation

We show that our framework is capable of generating designs that meet diverse latency targets and resource constraints. Figure 6 illustrates the latency variations using the HE parameter sets. We adjust $\mathcal{BW}_{max}$ between 128 GB/s and 460 GB/s. We gradually increase the DSP resource constraint ($\mathcal{D}_{max}$) to approach the maximum capacity of the U280 FPGA, which allows more ALUs (*NumAlu*) to be instantiated. We perform design space exploration to determine the values of the design parameters and implement the design on the target FPGA. All the designs achieve the target frequency of 250 MHz. Our DSE can estimate the latency of the designs with a very small error margin (less than 5%). The prediction error is primarily due to not modeling the initial startup and completion overhead. The generated designs that achieve the lowest latency are shown in Figure 6. Our framework can support a wide spectrum of latency by adjusting the DSP resources. Note, when *NumAlu* is a small value, HE operations become compute-bound and shows less sensitivity to changes in $\mathcal{BW}_{max}$. CtCtAdd performs limb element-wise operation with no data reuse. Its latency is proportional to *NumAlu* or

$\mathcal{BW}_{max}$, whichever is lower. CtCtMult and Rotate have a mixture of memory-bound and compute-bound subroutines. Therefore, high bandwidth and a large number of ALUs are both effective ways to reduce latency. Rescale is less sensitive to memory bandwidth because its latency is largely determined by the compute intensive Ntt and Intt subroutines.

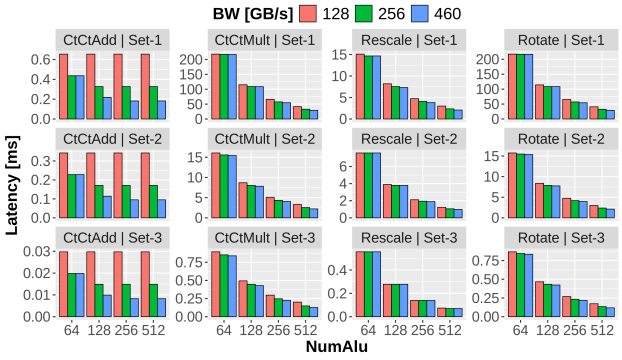

Fig. 6. Latency with respect to *NumAlu* and $\mathcal{BW}_{max}$.

Figure 7 shows the execution time breakdown for CtCtMult, with $NumAlu = 256$ and $\mathcal{BW}_{max} = 460$. ModUp, InnerProd, and ModDown together account for over 85% of the total execution time. Set-1 ($\alpha = 1$) and Set-2 ($\alpha = 8$) require a much larger number of limbs to be generated or reduced than Set-3 ($\alpha = 9$). Therefore ModUp and ModDown take a significant portion of the runtime. Rotate has a similar breakdown than CtCtMult. The execution time of Rescale is mostly taken up by Ntt and Intt. All other HE operations (e.g., CtCtAdd) only involve limb element-wise computations.

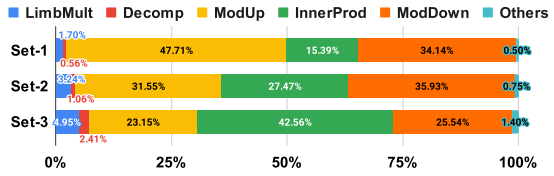

Fig. 7. Breakdown of the CtCtMult benchmark.

DSE Evaluation: We evaluate the effectiveness of the DSE in the proposed framework. As discussed in previous sections, on-chip SRAMs are shared resources between the permute pipeline and the scratchpad. Allocating the maximum DSP resources is unnecessary if other modules are the bottleneck. The DSE can automatically identify design parameters for low latency implementation of HE operations. Figure 8 illustrates the possible designs for CtCtMult under parameter Set-1. The x-axis shows the estimated latency from the DSE, the y-axis shows *PermTput*, and each $\times$ mark represents a design point evaluated by the DSE. Designs closer to the bottom left are better. The DSE explored a wide variety of design points and constructed a Pareto curve for latency. Clustering regions (e.g., design points in ❶) indicate that the Permute Pipeline is the bottleneck, therefore maximizing other design parameters does not significantly impact latency. In some cases, *NumAlu* is the bottleneck, so changing the values of *PermTput* has minimal effect (e.g., design points in ❷).

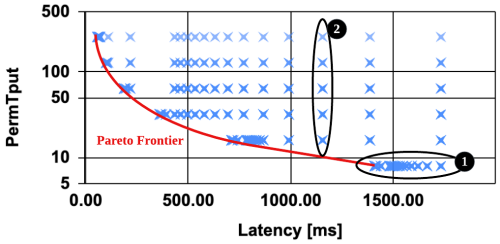

Fig. 8. DSE of accelerator generation for CtCtMult benchmark.

C. Evaluation of Framework Optimizations

1) Benefits of Sharing Computation Datapaths: Previous works such as FxHENN and Poseidon [14], [15] allocate dedicated hardware resources for NTT/INTT and other HE computations respectively, resulting in underutilized compute resources and increased latency. Our layered approach enables all subroutines of HE operations to execute on the same modular ALUs and the permute pipeline. In Figure 9, we compare the compute utilization of our design with that of Poseidon [15]. Compute utilization is defined as the ratio of active DSPs to the total DSPs in the design during the execution of an HE subroutine. While Poseidon achieves up to 82% utilization for NTT/INTT and 53% for other HE computations, our approach reaches 100% utilization..

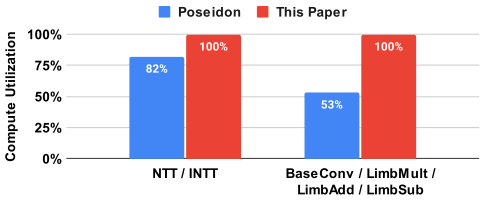

Fig. 9. Compute utilization comparison between Poseidon and our work.

2) Benefits of HE Subroutine Fusion: Figure 10 illustrates the speedup of fusing HE subroutines compared to no fusion. In this comparison, we assess the execution times of different HE operations under two I/O bandwidth constraints $\mathcal{BW}_{max}$: 128 GB/s and 460 GB/s. The fusion technique is beneficial for HE operations that involve consecutive Limb Dataflow subroutines. As the available I/O bandwidth decreases, more subroutines become memory bound, benefiting more from HE subroutine fusion. We observe up to $1.8\times$ speedup with HE subroutine fusion.

D. Resource Consumption

Table III shows the resource utilization of a sample design for CtCtMult. It is implemented using $NumAlu = 256$, $PermTput = 256$, $ScratchSize = 16$ MiB. The modular ALUs primarily use DSP resources, with each ALU requiring 12 DSPs. The scratchpad memory is divided into four banks, with each bank consists of 128 URAM instances. The SPN uses

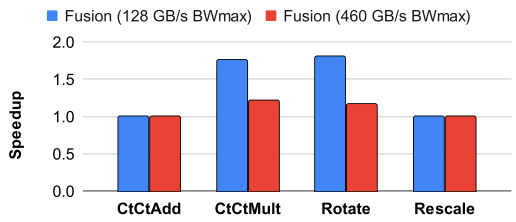

Fig. 10. Speedup on HE subroutine fusion on parameter Set-2.

BRAM resources for its temporal permutation operations. We observe that the consumption of LUT and FF is relatively low compared to the available resources.

TABLE III
FPGA RESOURCE USAGE OF A GENERATED DESIGN.

Work	DSP	URAM	BRAM	LUT	FF
Modular ALUs	3072	0	0	286976	489728
Permute Pipeline	0	0	256	102504	195160
Scratchpad	0	512	0	57696	90304
Others	0	0	0	153756	253708
Total	3072 [34%]	512 [54%]	256 [18%]	600932 [47%]	1028900 [40%]

E. Comparison with State-of-the-art

We compare the designs generated by our framework with state-of-the-art CPU, GPU and FPGA designs [24], [15], [26]. We use the same set of HE parameters as reported by the prior works. We use our framework to generate designs on U280 FPGA. The CPU baseline is measured on Intel Xeon Gold 6234 CPU using a single thread. The GPU baseline uses an NVIDIA Tesla V100 GPU. The GPU platform has more than $2\times$ peak HBM bandwidth than the FPGA platform used in our experiments. Table IV shows the CPU and GPU latency comparison results (in ms) on various HE operations. The PtCtMult benchmark is purely memory-bound, therefore the GPU design outperforms our design due to its higher HBM bandwidth. Benefiting from FPGA's application-specific logic (e.g., highly efficient modular ALUs), our generated designs achieve up to $1.3\times$ speedup over the GPU baselines and over $215\times$ speedup compared to CPU baselines.

TABLE IV
LATENCY COMPARISON WITH CPU AND GPU WORKS.

Work	Freq [GHz]	PtCtMult Lat. [ms]	CtCtMult Lat. [ms]	Rotate Lat. [ms]
CPU [24]	3.3	26.2	2631.6	2564.1
GPU [24]	1.3	0.18	17.4	16.8
This paper	0.25	0.29	12.6	11.9

TABLE V
LATENCY COMPARISON WITH FPGA WORKS.

Work	Freq [GHz]	PtCtMult Lat. [ms]	CtCtMult Lat. [ms]	Rotate Lat. [ms]
HEAX [26][1]	0.3	0.2	8.4	-
Poseidon [15][1]	0.45	0.08	3.7	3.3
This paper	0.25	0.08	4.2	3.9

[1] HEAX and Poseidon are optimized for specific HE parameters, whereas our framework supports a broader range of HE parameters.

Table V shows the latency (ms) comparison against state-of-the-art FPGA baselines [26], [15]. HEAX and Poseidon are FPGA accelerators tailored for a *fixed* set of HE parameters. This allows them to incorporate manual implementation optimizations for achieving high FPGA frequencies. In all the benchmarks, our designs demonstrate superior performance compared to HEAX. While Poseidon shows better performance due to its higher FPGA frequency (which is $1.8\times$ faster than ours), our design proves to be more efficient, requiring fewer cycles (up to $1.58\times$) to complete the benchmarks. This efficiency is primarily attributed to our approach of sharing computation datapaths and implementing dataflow optimizations.

VII. COMPOSING END-TO-END HE APPLICATIONS

We demonstrate our framework's ability to support end-to-end HE applications using HE Convolution Neural Network (CNN) inference [36], [19], [23], [22]. In HE CNN, linear operations, such as matrix multiplication and convolution, can be broken down into sequences of HE operations defined in our API layer [36], [37]. Non-linear operations are approximated with high-degree polynomials [38], [39], [40]. An entire HE CNN inference can be represented as a dependency graph, with each node corresponding to an HE operation. This dependency graph is the input to our framework. Our framework performs a topological sort on the graph to determine the processing order of the nodes in the generated design.

HE CNN Inference Benchmarks: Several previous studies have developed HE CNN models for privacy-preserving inference [19], [22], [23], [33]. In this work, we evaluate two HE CNN models from LoLa [33]. Table VI lists the model architecture and computation complexity. We choose $N = 8192$ and $N = 16384$ for the MNIST and Cifar10 models respectively based on prior works [14], [33]. We follow the same activations and parameters packing scheme as Lola.

TABLE VI
HE CNN BENCHMARKS.

Networks	Layers	Num HE Ops	Model Size [MB]
MNIST	Cnv1, Act1, Fc1, Act2, Fc2	0.83K	15.57
Cifar10	Cnv1, Act1, Cnv2, Act2, Fc2	82.73K	2471.25

Table VII and Table VIII summarize the comparison results. Lola performs a similar number of HE operations as our implementation. The difference in the number of HE operations is due to Lola's use of the BFV scheme [17], which differs from the CKKS scheme we employ. Falcon [41] reduces HE operations by applying homomorphic DFT in convolutional and fully connected layers. Both Lola and Falcon run on an Azure standard B8ms virtual machine with 8 vCPUs. Compared to CPU implementations, our designs achieve up to $60\times$ improvement. FxHENN [14] targets small-scale FPGAs. We match the FPGA resource constraints to the ACU15EG FPGA used by FxHENN to generate accelerators for the two benchmarks. FxHENN allocates dedicated hardware resources

to HE subroutines, resulting in longer latency. Our designs outperform FxHENN by up to $1.58\times$.

TABLE VII
COMPARISON OF HE CNN INFERENCE ON MNIST.

Work	Platform	Num HE Ops	λ	N	Q	Lat. [s]
Lola [33]	8× vCPUs	798	128	14	440	2.2
Falcon [41]	8× vCPUs	626	128	14	440	1.2
FxHENN [14]	ACU15EG	826	128	13	210	0.19
This paper	ACU15EG	826	128	13	210	0.12
This paper	U280	826	128	13	210	0.02

TABLE VIII
COMPARISON OF HE CNN INFERENCE ON CIFAR10.

Work	Platform	Num HE Ops	λ	N	Q	Lat. [s]
Lola [33]	8× vCPUs	123K	128	14	440	730
Falcon [41]	8× vCPUs	21K	128	14	440	107
FxHENN [14]	ACU15EG	82K	192	14	252	54.1
This paper	ACU15EG	82K	192	14	252	42.3
This paper	U280	82K	192	14	252	7.2

VIII. RELATED WORK

CPUs/GPUs. Several libraries [42], [43], [44] have been developed to enable efficient execution of HE operations on CPUs. Gazelle [36] proposed low overhead packing approaches to accelerate HE linear algebra operations. However, CPU implementations of HE still suffer poor performance due to the limited computing resources. Jung et al. [24] implemented GPU acceleration for various HE operations. Their approach focuses on GPU-specific operation and kernel fusions to reduce latency.

ASICs. F1 [34] is an ASIC designed for the BGV scheme. F1 only supports small HE parameters, which limits its practicality. CraterLake [35] enables unlimited HE operation by supporting bootstrapping. ARK [45] mitigates the memory bandwidth bottleneck by generating the key switching keys on-the-fly instead of loading them from DRAMs/HBMs. BTS [46] optimizes bootstrapping for a fixed set of HE parameters and designing a customized ASIC tailored to these parameters. While ASIC-based HE accelerators deliver high performance, they require substantial hardware resources, such as hundreds of megabytes of on-chip SRAMs (BTS, ARK).

FPGAs. Existing works either do not support all the HE operations [26], [30], [47] or are restricted to fixed HE parameters [25]. Yang et al. [48] proposed an FPGA accelerator for HE matrix-vector multiplication. They developed techniques to reduce the latency of rotation through exploiting on-chip limb reuse. The architecture of the design is tailored to a specific FPGA device, limiting its applicability to devices with different resource characteristics. Poseidon [15] proposed an FPGA accelerator for HE operations based on a high throughput implementation of Ntt and Automorph. FxHENN [14] proposed an HLS-based framework for executing HE operations on FPGAs. Both Poseidon and FxHENN assign dedicated FPGA resources to different HE subroutines. Their approaches lead to hardware resource duplications and compute underutilization, particularly when the subroutines are not executed simultaneously in an HE operation.

IX. CONCLUSION

In this paper, we presented a framework for generating low latency FPGA implementations of HE operations. The framework takes HE parameters and FPGA resource constraints as input and outputs synthesizable Verilog code. We proposed a layered approach combined with a parameterized HE domain specific architecture to efficiently reuse computation datapaths across different subroutines within an HE operation and improve compute utilization. We developed a design space exploration tool for automatically determining the optimal design parameters. Performance evaluations show that our framework significantly improves latency compared to state-of-the-art CPU implementations for various HE operations by up to $215\times$.

X. ACKNOWLEDGEMENT

This work has been sponsored by the U.S. National Science Foundation (NSF) under grants SaTC-2104264 and CSSI-2311870. Equipment by AMD AECG is greatly appreciated. Distribution Statement A: Approved for public release. Distribution is unlimited.

REFERENCES

[1] J. Kim, C. Lee, H. Shim, J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Encrypting controller using fully homomorphic encryption for security of cyber-physical systems," *IFAC-PapersOnLine*, vol. 49, no. 22, pp. 175–180, 2016.

[2] D. Archer, L. Chen, J. H. Cheon, R. Gilad-Bachrach, R. A. Hallman, Z. Huang, X. Jiang, R. Kumaresan, B. A. Malin, H. Sofia *et al.*, "Applications of homomorphic encryption," in *Crypto Standardization Workshop, Microsoft Research*, vol. 14. sn, 2017.

[3] O. Kocabas and T. Soyata, "Towards privacy-preserving medical cloud computing using homomorphic encryption," in *Virtual and Mobile Healthcare: Breakthroughs in Research and Practice*. IGI Global, 2020, pp. 93–125.

[4] Ö. Kocabaş and T. Soyata, "Medical data analytics in the cloud using homomorphic encryption," in *E-Health and Telemedicine: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2016, pp. 751–768.

[5] O. Masters, H. Hunt, E. Steffinlongo, J. Crawford, F. Bergamaschi, M. E. D. Rosa, C. C. Quini, C. T. Alves, F. de Souza, and D. G. Ferreira, "Towards a homomorphic machine learning big data pipeline for the financial services sector," *Cryptology ePrint Archive*, 2019.

[6] M. Albrecht, M. Chase, H. Chen, and et al, "Homomorphic encryption security standard," Tech. Rep., 2018.

[7] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

[8] J. Hennessy and D. Patterson, "A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[9] Xilinx, "Xilinx UltraScale+ HBM FPGAs," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html, 2020.

[10] Intel, "Stratix 10 MX FPGAs," https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html, 2020.

[11] "Amazon f1 instance," https://aws.amazon.com/ec2/instance-types/f1/.

[12] Alibaba, "Alibaba Cloud," https://www.alibabacloud.com/.

[13] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 30–39.

[14] Y. Zhu, X. Wang, L. Ju, and S. Guo, "Fxhenn: Fpga-based acceleration framework for homomorphic encrypted cnn inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 896–907.

[15] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2023.

[16] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson Jr., Eds. Cham: Springer International Publishing, 2019, pp. 347–368.

[17] 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*, R. Avanzi and H. Heys, Eds., 2017.

[18] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12, 2012.

[19] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," ser. ICML'16.

[20] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*.

[21] B. Reagen and et al, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[22] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: towards deep learning over encrypted data," in *Annual Computer Security Applications Conference (ACSAC 2016), Los Angeles, California, USA*.

[23] A. QaisarAhmadAlBadawi, J. Chao, and et al, "Hcnn, the first homomorphic cnn on encrypted data with gpus," *IEEE Transactions on Emerging Topics in Computing*.

[24] 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.

[25] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2023.

[26] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," ser. ASPLOS '20.

[27] Xilinx, "Xilinx UltraScale+ FPGAs," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[28] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2003.

[29] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

[30] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[31] R. Chen and V. K. Prasanna, "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[32] Y. Yang, W. Long, R. Kannan, and V. K. Prasanna, "Fpga acceleration of rotation in homomorphic encryption using dynamic data layout," in *2023 IEEE Annual International Symposium on Field-Programmable Logic and Applications (FPL)*. IEEE, 2023.

[33] A. Brutzkus, R. Gilad Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[34] 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*. New York, NY, USA: Association for Computing Machinery, 2021.

[35] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[36] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1651–1669.

[37] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Banwidth efficient homomorphic encrypted matrix vector multiplication accelerator on fpga," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–9.

[38] F. Boemer and et al, "Ngraph-he2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, ser. WAHC'19, 2019.

[39] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[40] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Fpga accelerator for homomorphic encrypted sparse convolutional neural network inference," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2022, pp. 1–9.

[41] Q. Lou, W.-j. Lu, C. Hong, and L. Jiang, "Falcon: Fast spectral inference on encrypted data," *Advances in Neural Information Processing Systems*, vol. 33, pp. 2364–2374, 2020.

[42] "Microsoft SEAL (release 3.6)," https://github.com/Microsoft/SEAL, Nov. 2020, microsoft Research, Redmond, WA.

[43] S. Halevi and V. Shoup, "Design and implementation of helib: a homomorphic encryption library," *Cryptology ePrint Archive*, 2020.

[44] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63.

[45] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1237–1254.

[46] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," *arXiv preprint arXiv:2112.15479*, 2021.

[47] S. S. Roy, A. C. Mert, S. Kwon, Y. Shin, D. Yoo *et al.*, "Accelerator for computing on encrypted data," *Cryptology ePrint Archive*, 2021.

[48] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Banwidth efficient homomorphic encrypted matrix vector multiplication accelerator on fpga," in *2022 IEEE International Conference Field-Programmable Technology (FPT)*. IEEE, 2022.