2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

CROPHE: Cross-Operator Dataflow Optimization for Fully Homomorphic Encryption Accelerators

Xinhua Chen[†§*] Jiangbin Dong[¶*] Hongren Zheng[‡] Tian Tang[‡] Mingyu Gao[‡†]
Shanghai Qi Zhi Institute[†] Tsinghua University[‡] Fudan University[§] Xi'an Jiaotong University[¶]
xhchen21@m.fudan.edu.cn bill412@stu.xjtu.edu.cn zhenghr22@mails.tsinghua.edu.cn
tang-t21@mails.tsinghua.edu.cn gaomy@tsinghua.edu.cn

***Abstract*—Fully homomorphic encryption (FHE) enables the protection of data privacy at the cost of significantly higher computational demands. To alleviate its memory-bound bottlenecks, dataflow optimizations that maximize on-chip data reuse and minimize off-chip accesses could be leveraged. In this work, we exploit the opportunities of cross-operator dataflow optimizations in FHE accelerators, and propose a hardware-software co-design called CROPHE. On the hardware level, instead of overly-specialized functional units, CROPHE provisions a homogeneous and unified architecture that allows for flexible resource allocation and operator mapping. On the software level, the scheduling framework of CROPHE takes a comprehensive and systematic approach to explore various spatial and temporal data pipelining and sharing schemes across multiple operators, resulting in more efficient dataflow than prior work. We also propose novel cross-operator dataflow optimizations for the unique operators in FHE including number theoretic transforms and homomorphic rotations. The evaluation shows CROPHE significantly outperforms state-of-the-art designs by $1.77\times$ to $4.86\times$.**

***Index Terms*—homomorphic encryption, domain-specific accelerator, dataflow**

I. Introduction

With the rapid development of advanced machine learning algorithms (e.g., large language models) and their deployment on public cloud platforms, data privacy becomes a primary concern when users send their private data and queries to cloud servers. Privacy-preserving computing [17], [22], [38], [48] emerges as a new paradigm, which leverages modern cryptographic protocols to protect data during processing. Fully homomorphic encryption (FHE) is one powerful category of this kind [8], [9], [16], [20]. It can perform computations on encrypted ciphertext data, and the decrypted results would match those of the corresponding plaintext computations.

Despite the fascinating privacy-preserving capability, the Achilles' Heel of FHE is the excessive computational cost, in terms of both the much more complicated ciphertext operations and the largely expanded ciphertext data volumes compared to plaintext processing. Consequently, many domain-specific hardware accelerators for FHE have appeared [2], [3], [33]–[35], [49]–[51], [59]. They usually provision a large number of specialized functional units for each FHE operation, with abundant on-chip data buffers to avoid excessively accessing the slow and expensive off-chip memory.

Nevertheless, a critical issue of FHE algorithms, such as the widely used CKKS protocol [16], is their memory-bound nature [2]. Instead of relying on huge on-chip buffers and expensive high-bandwidth off-chip memories, a more cost-efficient approach is to optimize the dataflow scheduling of FHE accelerators, in order to maximize their on-chip data reuse and minimize the off-chip data accesses. Dataflow optimizations have been extensively studied and demonstrated successful for deep neural network accelerators [10], [12], [21], [37], [42], [55], [58]. In FHE algorithms, we find that rather than intra-operator dataflow that mainly relies on loop tiling and reordering within an operator, efficiently reusing data across different operators is more critical, because there are only few (e.g., two or three) loop dimensions in each operator, but the dependencies between operators are complex. Such *cross-operator dataflow* techniques could include forwarding the intermediate ciphertext data between adjacent operators without spilling to off-chip, and sharing the commonly needed auxiliary constant data among multiple co-running operators of the same type.

While some prior designs [2], [35] have preliminarily started to borrow the idea of cross-operator dataflow scheduling from deep neural networks to FHE, their strategies still face several inefficiencies. On one hard, their optimizations *lack a systematic perspective*. They are limited to simple cases (e.g., a few manually-designed operator fusion schemes), and only handle the intermediate ciphertext data but leave the other auxiliary data uncovered. Moreover, they also lack effective support for the unique operators in FHE, such as number theoretic transforms (NTTs) and automorphisms. On the other hand, existing designs all use several types of functional units that are *overly specialized to individual FHE operators*, with a fixed ratio of each type on the chip. This rigid architecture cannot efficiently support cross-operator dataflow schemes that require co-running various operators together on-chip. When the scheduled FHE operators mismatch with the fixed ratio of the accelerator's functional units, some hardware resources would be left underutilized with degraded performance.

In this work, we propose CROPHE, a *hardware-software co-design* approach to exploit cross-operator data reuse with optimized dataflow scheduling on FHE accelerators. On the hardware level, CROPHE provisions a set of *homogeneous and unified* processing elements (PEs), each of which can be used to execute any type of FHE operator, without over-specialization. Such flexibility of resource allocation and operator mapping allows us to schedule arbitrary kinds and

[*]Both authors contributed equally to this research.

amounts of operators to co-run on the chip while still maintaining high utilization. This is necessary when supporting cross-operator dataflow which involves many operators.

On the software level, CROPHE presents a comprehensive hierarchical cross-operator dataflow design space. For the intermediate ciphertext data between adjacent operators, we support *pipelining* them from the producer to the consumer; for the auxiliary constant data such as homomorphic evaluation keys, we allow *sharing* them on-chip if they are needed by multiple operators that are currently co-running. Such pipelining and sharing can be realized either in a spatial manner on multiple nearby hardware PEs, or in a temporal way on the same PE. Furthermore, the key benefit is the ability to pipeline/share *in a fine granularity* to save the required on-chip buffer space to hold these reused data.

In addition, we also propose two novel dataflow optimizations that allow for pipelining/sharing of more operators in finer granularities. The first one targets the NTT operation, leveraging the widely known *decomposition* method in a novel way to reduce the occurrences of data access pattern changes, which would otherwise break fine-grained pipelining. The second one targets the sharing of evaluation keys in homomorphic rotations, combining two previous proposals [2], [34] into a more balanced *hybrid* scheme, which also creates cross-operator sharing opportunities that are previously absent.

We evaluate the optimized dataflow schemes systematically discovered by CROPHE against several state-of-the-art accelerator designs [33]–[35], [51] and dataflow techniques [2]. With similar area budgets, CROPHE outperforms the best among the baselines by $1.77\times$ to $4.86\times$ on several typical FHE workloads. The benefits of CROPHE would increase if we further shrink the on-chip SRAM capacity. These performance improvements are mainly from the significant reduction of data accesses to the on-chip global buffers and the off-chip memory, which in turn results from the increased data reuse in the corresponding cheaper levels of the memory hierarchy.

II. Background and Related Work

In this section, we describe the fully homomorphic encryption (FHE) algorithms, and summarize the recent hardware accelerator designs to alleviate their performance challenges.

A. Fully Homomorphic Encryption (FHE)

Common FHE schemes include BGV [8], B/FV [9], [20], CKKS [16], etc. While the former two focus on integer data, CKKS enables fixed-point arithmetics that are particularly important for privacy-preserving data analysis applications, e.g., private inference on deep neural networks (DNNs) [17], [22], [38], [48]. Its approximate nature also enables much better performance. Thus we choose CKKS as the state-of-the-art FHE representative, but our techniques are also conceptually applicable to other schemes.

Ciphertext format. In CKKS, a vector of values is first encoded into a plaintext, and then encrypted into a ciphertext $\mathsf{ct} = (b(X), a(X))$ that consists of two polynomials in $\mathbb{Z}_Q[X]/(X^N+1)$. Each of such polynomials has N coefficients in $\mathbb{Z}_Q$ (integers modulo Q). N is typically a power of 2 between 2^{14} and 2^{18} [35], and Q should be thousands of bits long, both to meet the required security level. To avoid expensive arithmetics on wide integers in $\mathbb{Z}_Q$, a standard optimization is to leverage the Chinese Remainder Theorem to construct a *Residue Number System* (RNS) [4], [5], [15]. Specifically, the large modulus Q is decomposed into the product of $\ell+1$ smaller (e.g., less than 64 bits) prime numbers $q_0 q_1 \ldots q_\ell$ called the *RNS bases*, and correspondingly a number in $\mathbb{Z}_Q$ can be represented by $\ell+1$ smaller numbers in $\mathbb{Z}_{q_0} \times \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_\ell}$. Moreover, additions and multiplications in $\mathbb{Z}_Q$ become element-wise additions and multiplications in the RNS domain. In this way, each of $b(X)$ and $a(X)$ is decomposed to $\ell+1$ *limb polynomials* each with N coefficients, and can be viewed as an $(\ell+1)\times N$ matrix.

Homomorphic operations. CKKS allows certain homomorphic operations performed on ciphertexts, whose outputs, after decryption, equal to the corresponding plaintext operation results. For example, HAdd (homomorphic addition) adds the polynomials $\mathsf{ct}_{\text{add}} = \mathsf{ct}_0 + \mathsf{ct}_1 = (b_0(X) + b_1(X), a_0(X) + a_1(X))$. HMult (homomorphic multiplication) first does a tensor product between ct_0 and ct_1

$$\begin{aligned}\mathsf{ct}_0 \times \mathsf{ct}_1 &= (d_0(X), d_1(X), d_2(X)) \\ &= (b_0(X)\cdot b_1(X), a_0(X)\cdot b_1(X) + b_0(X)\cdot a_1(X), a_0(X)\cdot a_1(X))\end{aligned}$$

Then with the help of an *evaluation key* evk, a *key-switching* primitive converts the above result into a ciphertext $\mathsf{ct}_{\text{mult}} = (d_0(X), d_1(X)) + \text{KeySwitch}(d_2(X))$, which can be decrypted by the original secret key.

$$\text{KeySwitch}(d(X)) = P^{-1}(d(X)\cdot \mathsf{evk})) \tag{1}$$

The polynomial multiplications in HMult can be optimized using *number theoretic transforms* (NTTs) with $O(N \log N)$ complexity. Similar to a fast Fourier transform (FFT), an NTT converts a polynomial $a(X)$ from the *coefficient representation* to the *NTT representation* $\tilde{a}(X)$, using a set of constant *twiddle factors*. Then, a polynomial multiplication in the coefficient representation becomes an element-wise multiplication in the NTT representation. Usually the polynomials in FHE processing are kept in the NTT representation for efficient additions and multiplications.

The key-switching flow is shown in Figure 1. The evaluation key evk consists of two polynomials in $\mathbb{Z}_{PQ}/(X^N+1)$ each of shape $(k+\ell+1)\times N$, where P is a special modulus decomposed to a set of *extended RNS bases* $p_0 p_1 \ldots p_{k-1}$. Therefore, in Equation (1), d_2 must be first converted from $\mathbb{Z}_Q$ to $\mathbb{Z}_{PQ}$ through a process called *base conversion* (BConv), then multiplied with evk, and finally converted from $\mathbb{Z}_{PQ}$ back to $\mathbb{Z}_Q$ with BConv again. BConv requires data to be converted back to the coefficient representation through the inverse-NTT (iNTT) primitive. This results in a typical sequence of iNTT $\rightarrow$ BConv $\rightarrow$ NTT as shown in Figure 1.

To ensure sufficient security, the special modulus P must be large enough compared to the ciphertext modulus Q, resulting in high computation and storage cost. A critical optimization is to *decompose* ct and evk into $\beta = \lceil(\ell+1)/\alpha\rceil$

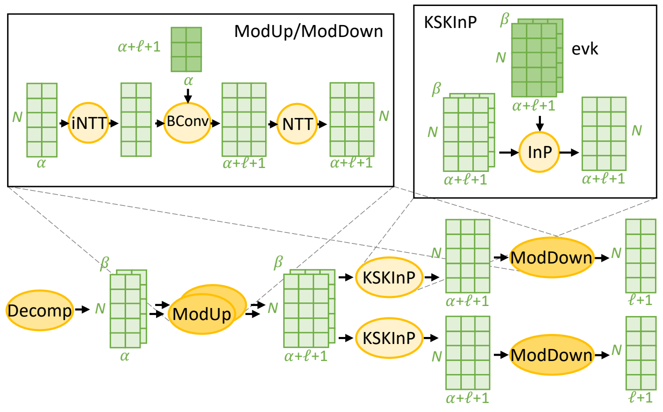
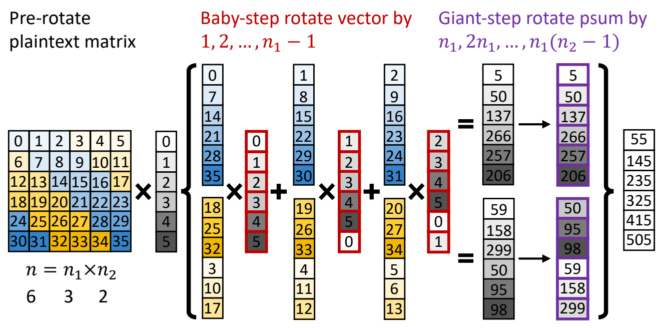


Fig. 1. Computation and data patterns of key-switching.



Pre-rotate plaintext matrix
Baby-step rotate vector by 1, 2, …, $n_1 - 1$
Giant-step rotate psum by $n_1, 2n_1, \dots, n_1(n_2 - 1)$
$n = n_1 \times n_2$
6 3 2

Fig. 2. An example of PtMatVecMult using BSGS.

digits, each corresponding to α limbs [25]. Then P only needs to be larger than the product of the α RNS bases q_i's within each digit rather than all q_i's, i.e., setting $k = \alpha$ when p_j's and q_i's are close. This makes the evk shape become $2 \times \mathsf{dnum} \times (\alpha + \ell + 1) \times N$, in which dnum is the maximum β for $\ell_{\max} = L$. BConv is now applied to each of the β digits independently, multiplying each $\alpha \times N$ matrix with a constant matrix to obtain a result matrix of $(\alpha + \ell + 1) \times N$. The multiplication with evk is then an inner-product between two $\beta \times (\alpha + \ell + 1) \times N$ tensors reducing along the β dimension. In summary, key-switching involves multiple steps: digit decomposition (Decomp), BConv to wider fields (ModUp), inner-product with evk (KSKInP), and BConv back (ModDown).

Besides the above HAdd and HMult that work on two ciphertexts, CAdd/CMult and PAdd/PMult process a ciphertext with a constant scalar and a plaintext, respectively, using simple element-wise operations. Also, HRot (homomorphic rotation) circularly rotates the plaintext values across the vector slots in a ciphertext. It consists of two steps. An *automorphism* first permutes the coefficients of polynomial $a(X)$ with the index mapping $i \to i \cdot 5^r$, where r is the rotation amount. The result is denoted as $a(X^{5^r})$. Then a key-switching as in Equation (1) is applied to obtain $\mathsf{ct}_{\mathrm{rot}(r)} = (b(X^{5^r}), 0) + \mathrm{KeySwitch}(a(X^{5^r}))$. Note that the HRot evk is different from that of HMult. Actually, each rotation amount r requires a different evk.

Multiplicative level and bootstrapping. Unfortunately, the above homomorphic operations, especially CMult, PMult, and HMult, would increase the noise in the ciphertext and eventually make it undecryptable. To mitigate this issue, CKKS scales the ciphertext down after each multiplication to keep the noise error tolerable. With RNS, such scale-down can be efficiently realized as dividing the ciphertext by the last prime modulus q_ℓ and removing it from the RNS bases, until there is only one modulus q_0, at which point we have to "refresh" the ciphertext with a special procedure called *bootstrapping*. We call the current number of ciphertext moduli ℓ as the (multiplicative) *level*, and the maximum level is L. Essentially, bootstrapping homomorphically transforms a ciphertext under q_0 into a new ciphertext under $\Pi_{\ell=0}^{L-L_{\mathrm{boot}}} q_\ell$. Note that bootstrapping itself contains a number of multiplications (see below), and consumes a certain number of levels L_{boot}.

The major parts of bootstrapping [7], [14], [25] transform the coefficients of the plaintext polynomial to the ciphertext vector slots (CoeffToSlot), homomorphically evaluate a modular reduction (EvalMod), and then transform the coefficients back (SlotToCoeff). Both CoeffToSlot and SlotToCoeff involve plaintext matrix-vector multiplications (PtMatVecMult), and EvalMod uses a series of HMult and CMult operations to calculate a high-degree polynomial function.

The PtMatVecMult kernel can be calculated in an optimized way called *baby-step giant-step* (BSGS) [23], as shown in Algorithm 1 and Figure 2. Essentially, for an $n \times n$ plaintext matrix, we split n into $n_1 \times n_2$ where $n_1, n_2 \sim \sqrt{n}$. This reduces the number of rotations from $O(n)$ to $O(\sqrt{n})$.

Algorithm 1: PtMatVecMult using BSGS.

```
Input: a ciphertext ct encrypting a vector of length $n$, an
       plaintext matrix $\mathbf{M}$ of size $n \times n$ and $n = n_1 \times n_2$.
Output: $\mathsf{ct}' = \mathbf{M} \times \mathsf{ct}$.
1 for $i \leftarrow 1$ to $n_1 - 1$ do
2     $\mathsf{ct}_i = \mathrm{HRot}_i(\mathsf{ct})$;                      // baby-step rot
3 $\mathsf{ct}' \leftarrow (0, 0)$;
4 for $j \leftarrow 0$ to $n_2 - 1$ do
5     $\mathsf{r} \leftarrow (0, 0)$;                                     // psum
6     for $i \leftarrow 0$ to $n_1 - 1$ do
7         $\mathsf{r} \leftarrow \mathrm{HAdd}(\mathsf{r}, \mathrm{PMult}(\mathsf{ct}_i, \mathrm{Rot}_{-n_1 \cdot j}(\mathbf{M}_{\mathrm{diag}}^{n_1 \cdot j + i})))$;
8     $\mathsf{ct}' \leftarrow \mathrm{HAdd}(\mathsf{ct}', \mathrm{HRot}_{n_1 \cdot j}(\mathsf{r}))$;        // giant-step rot
9 $\mathsf{ct}' \leftarrow \mathrm{HRescale}(\mathsf{ct}')$;
```

Summary. In short, CKKS consists of the following basic operators: element-wise tensor additions/multiplications, matrix/tensor multiplications (e.g., BConv, inner-product), NTTs, and automorphisms.

B. Hardware Acceleration for FHE

Modern FHE schemes commonly work on huge amounts of data (e.g., N up to 2^{18}), with each data element being a very wide integer number (e.g., $\log Q$ up to thousands of bits) or dozens of narrow numbers if decomposed. These characteristics put challenges on both computations and data accesses. The numerous wide modular arithmetic operations result in high computational demands; the large memory footprints of up to GB-level data cause memory access bottlenecks. Traditional CPUs have insufficient computation capabilities. GPUs, while being highly parallel, lack specialized modular arithmetic units as well as efficient memory hierarchy and data communication optimizations. They also exhibit the general-purpose overheads like register file access and instruction control cost. As a result, many specialized accelerators have been designed to boost the performance of FHE.

Early FHE accelerators [49], [54] were only designed for specific parameters such as N and L. F1 [50] was the first programmable FHE accelerator that supported flexible parameters. It adopted many functional units for wide vector modular arithmetics, as well as specialized units for NTT and automorphism. BTS [35] and CraterLake [51] further enabled efficient bootstrapping support to scale to more complex FHE workloads. They introduced new hardware units and dataflow schemes for key primitives like BConv, NTT, and automorphism. ARK [34] mainly focused on the memory bottleneck due to large amounts of off-chip data accesses. It proposed algorithm optimizations to improve data reuse and trade extra computations for fewer data accesses. SHARP [33] exploited the impact of machine word size choices and concluded that 36-bit words achieve the best balance. It also used a hierarchical microarchitecture together with careful data scheduling to reduce the on-chip buffer capacity. BitPacker [52], on the other hand, proposed algorithmic enhancements to CKKS to more densely pack data into machine words, reducing unused bits that waste datapath and storage resources. MAD [2] focused on the memory bottleneck of FHE workloads, and proposed a series of both hardware (e.g., data caching, operation fusion) and algorithm (e.g., operation merging, hoisting) optimizations to reduce off-chip memory accesses.

Poseidon [59] and FAB [3] were two FPGA-based FHE accelerators. The former decomposed key homomorphic operations into low-level primitives in order to share hardware functional units; the latter proposed specific schemes to fit data within limited FPGA on-chip SRAM and also explored multi-FPGA scale-out solutions. There also have been accelerator designs that support the new torus FHE (TFHE) protocols [1], [18], [44], [45], [61], which are orthogonal to our work.

III. Cross-Operator Scheduling

A. Motivation

With the excessive volumes of ciphertexts and auxiliary data such as evaluation keys and BConv constant matrices, many FHE operators are *highly memory-intensive*, with low compute-to-data ratios as demonstrated in prior work [2]. To alleviate the memory bottleneck, an efficient approach is to optimize the *dataflow scheduling* methods to maximize on-chip data reuse and minimize off-chip memory accesses. Such an approach has already been demonstrated successful for deep neural network (DNN) accelerators, with extensive studies exploring the rich design spaces of both intra-operator loop transformations [12], [27], [29], [37], [39], [42], [58] and cross-operator data forwarding [10], [21], [55].

When borrowing this idea from the DNN accelerators to the FHE domain, we argue that we should primarily focus on *cross-operator data reuse* rather than intra-operator reuse. This is because the reuse patterns within most individual FHE operators are relatively simple. For instance, element-wise operators are streaming with no data reuse. For BConv, the constant matrix of size $(\alpha+\ell+1)\times\alpha$ is small enough (e.g., <1000 elements for α and ℓ around 10s) to easily fit into the on-chip buffers. The inner product with evk only reduces along β, so it can stream in each chunk of β elements, following the $O(\beta)$ caching technique in MAD [2] to fully reuse all data.

In contrast, cross-operator dataflow is more critical in FHE. On one hand, a typical FHE program could involve many FHE operators with complex producer-consumer dependencies between them, making data reuse across operators non-trivial to optimize. On the other hand, the cross-operator intermediate ciphertext data volumes are significant compared to the mostly simple operators with no (e.g., element-wise) or small amounts (e.g., BConv) of other constant data. Even for the large evk, previous designs like SHARP have observed that the working set size is more sensitive to the number of temporary ciphertexts than the evk [33].

Limited dataflow techniques. Several designs have preliminarily exploited cross-operator data reuse. For example, BTS adopted a specific timing schedule to execute the operators in an HMult [35]. MAD leveraged operator fusion to efficiently cache the intermediate data across adjacent operators [2]. However *these existing designs are rather limited and lack systematic methodologies*. First, their schemes are restricted to a few special cases. The BTS schedule is only for HMult; and MAD only proposes a few individual manually-designed optimizations. Second, the existing operator fusion techniques in MAD mainly target the intermediate ciphertexts, but not the constant auxiliary data like evks, leaving ample opportunities unexploited. Third, their designs only support regular computations. When encountering operators that exhibit complex data permutations, e.g., NTT and automorphism, operator fusion must terminate, which reduces the data reuse scope. To overcome these issues, we aim to propose a more comprehensive framework for more systematic dataflow scheduling.

Overly specialized hardware resources. Furthermore, to maximally reuse data across operators, it is desired to be able to *flexibly group various numbers of operators for co-running on-chip*. With a fixed on-chip buffer capacity, operators with few data to cache can form a larger group to simultaneously reside on-chip, while we can only fit a few operators if each needs massive data. This in turn requires us to flexibly allocate the on-chip compute resources to the various number of operators. Unfortunately, most existing FHE accelerators use *overly specialized functional units* for each operator type, with a fixed ratio of different functional units provisioned on hardware. For example, almost all state-of-the-art accelerators consist of separate functional units for element-wise addition/multiplication, NTT, automorphism, and BConv [33]–[35], [50], [51]. MAD does not specify their own hardware architecture but follows the previous designs [2]. Even though Poseidon has lowered

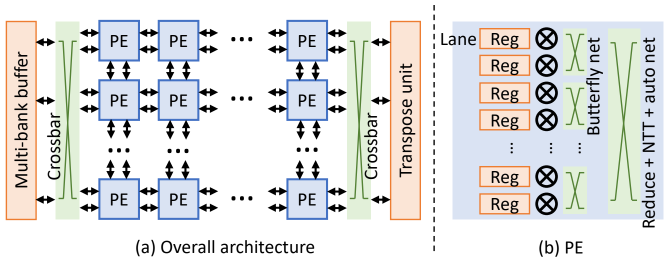


Fig. 3. Hardware architecture of CROPHE.

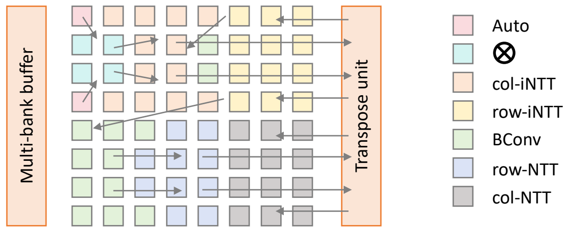


Fig. 4. An example of mapping the first spatial pipeline in Figure 6 to an 8×8 PE array in CROPHE. Arrows show data transfer directions.

its representation level, it still uses specialized NTT and automorphism units separate from the element-wise ones [59]. Such over-specialization would greatly limit the effectiveness of cross-operator scheduling. On one hand, the fused operator types must match the hardware functional units. If some operator types are missing, their hardware units would be left idle. On the other hand, with different data shapes and algorithm parameters, different operator groups have diverse computation demands for each type. It is unlikely that the varying computation demands always match the hardware resources, leading to resource underutilization.

B. Our Approaches

In this paper, we systematically study the *cross-operator dataflow scheduling* problem, and propose CROPHE, a comprehensive framework to thoroughly explore the design space. CROPHE is a hardware-software co-design. On the hardware level (Section IV), we use a set of homogeneous and unified processing elements (PEs), which are not overly specialized to specific FHE operators, but any PE can be used to map any operator. This allows for flexible resource allocations to co-run arbitrary groups of operators on the accelerator chip, enabling various cross-operator dataflow scheduling. On the software level (Section V), we support a comprehensive hierarchical cross-operator dataflow design space, with both temporal and spatial operator mappings in both pipelining and sharing manners, which support efficient cross-operator data reuse for both intermediate ciphertext polynomials and auxiliary constant data. We further propose two novel optimizations that allow us to group more operators for higher reuse opportunities.

IV. Hardware Architecture

We first introduce the hardware architecture of CROPHE, which mainly consists of a set of homogeneous and unified processing elements (PEs). The homogeneity allows for flexible resource allocation to various numbers of operators in a group, making the hardware efficient to support flexible cross-operator dataflow schemes discovered by CROPHE.

A. Hardware Structures

Figure 3(a) shows the overall architecture of the accelerator chip. The central part is a 2D array of processing elements (PEs), connected in a mesh network-on-chip (NoC). The PEs can access the shared multi-bank SRAM buffers through a crossbar. On the other side of the chip, an SRAM-based transpose unit [33], [50] is provisioned to support fast on-chip data transposition, also connected to the PEs through a crossbar. It only needs a few MB capacity, much (e.g., $> 10\times$) smaller than the data buffers and thus incurring small area overheads. We optimize the mapping strategies to allow data transpose operators to use the transpose unit while reducing the data transfer distances on the NoC (Section IV-B).

Each PE follows the vector-style design and consists of multiple (e.g., 256) lanes of modular arithmetic units [19], as illustrated in Figure 3(b). Each lane contains one modular multiplier, a few modular adders/subtractors, and a small register file. We use Barrett reduction for modular arithmetics [6], similar to previous designs [33], [35]. Each pair of neighbor lanes can be combined to support the butterfly operation in NTTs [19]. The inter-lane network across all lanes consists of several stages of different networks, including a reduction tree, a constant-geometry network for NTTs [11], [43], and multiple levels of shift networks for automorphisms [19]. The area cost of these inter-lane network stages is small compared to the modular arithmetic units and register files.

The mesh NoC transfers data using lightweight packets, following the data producer-consumer dependencies that are statically determined by the operator mapping scheme. Specifically, for each PE that produces some output data, the PEs that consume these data to execute the next operators are statically known. Thus the NoC is able to route the data packets correctly. The communication between the PEs and the transpose unit is done similarly. Such a packet-based message passing scheme handles inter-PE communication explicitly. Also note that packets are transmitted hop-by-hop. Even the whole chip may have a large scale, the wire length per hop is not very long and will not become a clock frequency bottleneck. We further support the multicast functionality in the NoC [13], [28], [31], [53], which is useful for transferring data shared by multiple PEs (Section V-A). Switching between operator groups is fully synchronous; we wait for the completion of the previous group on all PE, and then start the next group of operators with the new mapping scheme. The above mechanisms are mature and widely used in DNN accelerators [10], [21], [55].

B. Operator Mapping

The key feature of our homogeneous hardware architecture is the ability to *support all the various operators in FHE on the unified PEs, so each PE can be flexibly allocated to any FHE operator*, with minimum resource idleness even when the cross-operator dataflow varies broadly. We now describe how to map each FHE operator to the PEs.

For element-wise operators like HAdd and PMult, they can be directly mapped to the multi-lane PEs in a vector style. For matrix multiplications like BConv, we further use the reduction tree in the inter-lane network of each PE, and the inter-PE NoC links, to reduce the partial sums after element-wise multiplications. For NTTs, we apply multi-dimensional decomposition [33], [35], [57], [60] to let the small sub-NTT size match the number of lanes in a PE, and use the transpose unit to conduct dimension transpositions. Each sub-NTT in one PE uses the PE pairs for butterfly operations, and the constant-geometry inter-lane network for data shuffles [11], [19], [43]. For automorphisms, we follow the prior design [19], which uses multiple shift stages in the inter-lane network to realize the required permutations.

CROPHE has two additional requirements. First, each operator may be allocated with varying amounts of hardware resources, depending on how many operators are co-running. We allocate resources *in the granularity of PEs*, i.e., all the lanes in a PE can be used by only one operator, not shared by others. Our scheduling framework in Section V determines the optimized number of PEs allocated to each operator. Specifically, to ensure all the co-running operators in a group have balanced throughput, we allocate their numbers of PEs proportional to their computational loads [21]. Once the resource allocation is decided, our mapping strategies described above can be easily extended to use the provisioned PEs. For example, assume each PE has 256 lanes. For an element-wise operator with $N = 2^{14}$, we can either use a single PE with 64 temporal iterations ($2^{14} = 256 \times 1 \times 64$), or use 16 PEs with only 4 iterations ($2^{14} = 256 \times 16 \times 4$). NTTs and automorphisms also support similar multi-dimensional decompositions [19], [35], [57] into the three levels of the intra-PE network, inter-PE NoC, and temporal iteration.

Second, when mapping operators to PEs, we would like to minimize the data transfer distances between adjacent operators. We generally follow the horizontal direction to map consecutive operators from left columns to right ones. When the number of PEs for an operator is small, we also allow mixing multiple operators in one column. For example, Figure 4 maps the first spatial group of operators in Figure 6. The first three operators, Auto, $\otimes$, and col-iNTT, follow the aforementioned pattern. A specific issue to optimize is the transpose operators, which must be executed on the rightmost transpose unit. We therefore map the operators after the transpose from right to left. In Figure 4, after col-iNTT, the data are directly transferred to the transpose unit, while the next row-iNTT receives the transposed result and sends its output to the left. When there are multiple (but usually no more than two) transpose operators, we split the PE array into horizontal bands, with the number of rows proportional to the computational demand of each segment. For example, Figure 4 has two transposes. In summary, the above mapping strategies are heuristic but already work well. We leave more optimized mapping as future work as it is not the focus of this work.

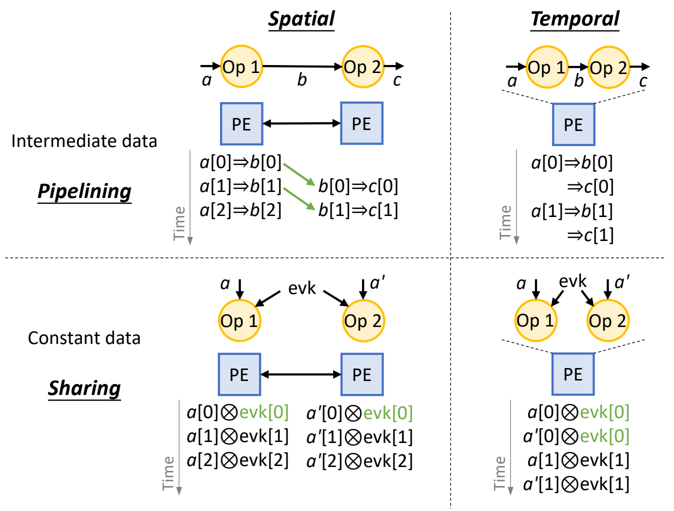


Fig. 5. Cross-operator dataflow taxonomy in CROPHE.

V. Scheduling Framework

In this section, we introduce the scheduling framework of CROPHE, which determines the cross-operator dataflow, i.e., which operators to group together for co-running, and how many PEs to allocate to each operator. We first illustrate a hierarchical taxonomy of cross-operator dataflow that is highly flexible and more comprehensive than previous ad-hoc optimizations [2], [35] (Section V-A). We next propose two novel optimizations, targeting NTT and HRot, respectively, to make them more friendly to cross-operator data reuse (Sections V-B and V-C). Then we describe our scheduling algorithm to identify optimized schemes for a given FHE workload (Section V-D).

Note that while the cross-operator dataflow scheduling is conceptually similar to those in DNN accelerators [10], [21], [55], to our best knowledge it is the first time to formally and systematically introduce such a framework for FHE accelerators with the specific operator types. In particular, NTT and HRot are unique in FHE and do not appear in DNNs. Our two optimizations for them are novel.

A. Cross-Operator Dataflow

CROPHE focuses on cross-operator scheduling that has multiple operators processed on-chip and shares data among them. There are two types of data, the intermediate ciphertext polynomials, and the auxiliary constant data such as evk and the BConv matrix. We discuss two techniques that reuse these two data types, respectively, as illustrated in Figure 5.

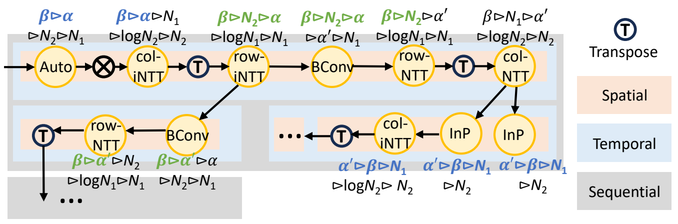
First, for intermediate polynomial data that are shared between their producer and consumer operators, we can forward them between these operators without spilling to off-chip. Depending on whether these operators are executed spatially or temporally, we define two schemes, similar to those in DNN accelerators [10], [21], [55] (Figure 5 top). First, multiple operators can be spatially processed by different PEs of the accelerator, with their intermediate data forwarded among the PEs. This is called *spatial pipelining*. In contrast, we can also temporally execute multiple operators one after another on the same set of PEs, forming *temporal pipelining*.

Second, for auxiliary constant data that are shared by operators of the same type, we can schedule multiple such operators to execute simultaneously on-chip. Then we can fetch the auxiliary data only once to feed all the operators. Similarly, we can also have *spatial sharing* and *temporal sharing* (Figure 5 bottom). In the spatial case, the fetched auxiliary data are forwarded to multiple simultaneously running operators; in the temporal case, the auxiliary data stay on-chip, and are used by multiple sequentially executed operators.

Fine-grained pipelining/sharing. The key benefit of both spatial and temporal pipelining and sharing is achieved when data are forwarded among operators *in a fine granularity* rather than as an entity [2], [21]. This can greatly reduce the on-chip buffer space requirement and save the chip area. As a simple example, assume two element-wise operators process a ciphertext polynomial consecutively. With a spatial pipeline, the second operator can start executing as soon as the first one forwards a single element, instead of waiting for the full $(\ell+1)\times N$ matrix. This is illustrated in Figure 5 top as the pipelining of $b[0], b[1], \ldots$. Similarly, with a temporal pipeline, the same PE processes each element of the matrix with the two operators in turn. In both cases, we only need an on-chip buffer keeping one element, and the full matrix can be streamed from the off-chip memory. Actually, the techniques of caching $O(1)$ and $O(\beta)$ limbs in MAD [2] are just special cases of such fine-grained pipelining. Figure 5 bottom further shows the streaming usage of $\mathsf{evk}[0], \mathsf{evk}[1], \ldots$, as examples for the fine-grained sharing of the constant auxiliary data.

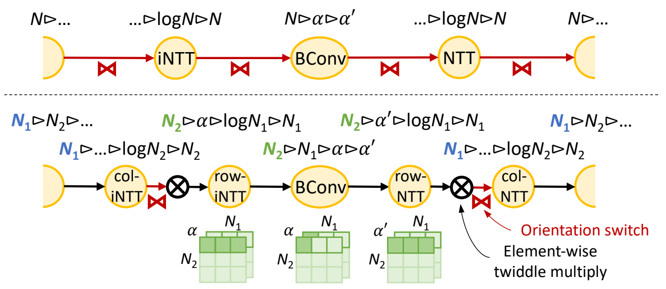
To enable data pipelining/sharing at a fine granularity, the co-running operators under pipelining/sharing must process each data granule in the same order. The nested loop structure is a common way to accurately describe how an operator processes its data, i.e., the operator's dataflow [42], [46], [58]. Most FHE operators have three loops, the slot dimension N, the limb dimension ℓ and α, and the digit dimension β. These loops can be further tiled (e.g., $N = N_1 \times N_2$) or reordered. For example, with a loop order $\ell \triangleright N$ (meaning outer $\triangleright$ inner), we process all the N slots of each limb before going to the next limb. Alternatively, an order $N_1 \triangleright \ell \triangleright N_2$ would process each limb in a granularity of N_2 slots. With this nested loop notation, fine-grained pipelining/sharing requires the operators to *have the same loops in the same order at the top few levels*, i.e., the outer iterations of the execution match between the operators. This condition ensures that these operators produce/consume each chunk of data in exactly the same order, so the data chunks can be pipelined through, without storing them all in the buffers. For example, as in Figure 6 bottom right, the two InP and the col-iNTT operators have the same top-level loops of $\alpha' \triangleright \beta \triangleright N_1$ (here $\alpha' = \alpha + \ell + 1$ is the limb count after ModUp). Therefore these operators stream every chunk of N_2 elements.

Hierarchical dataflow design space. The above spatial and temporal pipelining/sharing can be flexibly composed together. To simplify the dataflow design space while still retaining



$\beta \triangleright \alpha$ $\triangleright N_2 \triangleright N_1$
$\beta \triangleright \alpha \triangleright N_1$ $\triangleright \log N_2 \triangleright N_2$
$\beta \triangleright N_2 \triangleright \alpha$ $\triangleright \log N_1 \triangleright N_1$
$\beta \triangleright N_2 \triangleright \alpha$ $\triangleright \alpha' \triangleright N_1$
$\beta \triangleright N_2 \triangleright \alpha'$ $\triangleright \log N_1 \triangleright N_1$
$\beta \triangleright N_1 \triangleright \alpha'$ $\triangleright \log N_2 \triangleright N_2$
Auto
col-iNTT
row-iNTT
BConv
row-NTT
col-NTT
T
row-NTT
BConv
$\beta \triangleright \alpha' \triangleright N_2$ $\triangleright \log N_1 \triangleright N_1$
$\beta \triangleright \alpha' \triangleright \alpha$ $\triangleright N_2 \triangleright N_1$
...
col-iNTT
InP
InP
$\alpha' \triangleright \beta \triangleright N_1$ $\triangleright \log N_2 \triangleright N_2$
$\alpha' \triangleright \beta \triangleright N_1$ $\triangleright N_2$
$\alpha' \triangleright \beta \triangleright N_1$ $\triangleright N_2$
Transpose
Spatial
Temporal
Sequential

Fig. 6. A real-case example of the cross-operator dataflow scheme discovered by the CROPHE scheduler.



$N \triangleright \ldots$
$\ldots \triangleright \log N \triangleright N$
$N \triangleright \alpha \triangleright \alpha'$
$\ldots \triangleright \log N \triangleright N$
$N \triangleright \ldots$
iNTT
BConv
NTT
$N_1 \triangleright N_2 \triangleright \ldots$
$N_1 \triangleright \ldots \triangleright \log N_2 \triangleright N_2$
$N_2 \triangleright \alpha \triangleright \log N_1 \triangleright N_1$
$N_2 \triangleright N_1 \triangleright \alpha \triangleright \alpha'$
$N_2 \triangleright \alpha' \triangleright \log N_1 \triangleright N_1$
$N_1 \triangleright \ldots \triangleright \log N_2 \triangleright N_2$
$N_1 \triangleright N_2 \triangleright \ldots$
col-iNTT
row-iNTT
BConv
row-NTT
col-NTT
α
N_1
N_2
α'
Orientation switch
Element-wise twiddle multiply

Fig. 7. Use NTT decomposition to improve pipelining, reducing orientation switches by 2× in the sequence iNTT → BConv → NTT. Top: original. Bottom: decomposed.

sufficient flexibility, we focus on a hierarchical scheme. At the bottom level, we use spatial pipelining/sharing to fully utilize all the PEs in the accelerator. The operators added in each spatial group should share either intermediate or auxiliary data. Then, multiple spatial groups can form a temporal pipelining/sharing group, i.e., they reuse the same accelerator chip one after another. Note that either level can collapse; e.g., if all the bottom spatial pipelines only contain a single operator, these operators are simply a temporal group. Finally, for large accelerators, even without fine-grained pipelining/sharing, the buffer may still be able to fully cache the intermediate data. Thus we add a top level in which adjacent operators execute sequentially with their intermediate/auxiliary data fully materialized in the on-chip buffer. This results in a three-level hierarchy: sequential execution $\rightarrow$ temporal pipelining/sharing $\rightarrow$ spatial pipelining/sharing. Figure 6 illustrates an example of the optimized scheme for HRotate discovered by our scheduler, which exhibits the three levels.

Optimization goals and challenges. Recall that the goal of CROPHE is to maximize data reuse with minimum buffer space. This translates to adding as many operators into the pipelining/sharing groups while using as fine data granularities as possible. Our scheduling algorithm in Section V-D searches for optimized dataflow schemes towards this goal. Before that, we introduce two novel optimizations for intermediate ciphertext polynomial pipelining and auxiliary constant data sharing, respectively, which alleviate the bottlenecks when grouping corresponding operators.

B. Optimization: NTT Decomposition

When pipelining intermediate data, we need to have as many matched top-level loops as possible, with large loop sizes, to achieve the finest granularities. Among the FHE loops, N is usually significantly larger than the numbers of limbs and digits, so we would like to keep the N loop or its tiled subloops at the top level. This is easy for certain operators like element-wise, BConv, and evk inner-product. But NTT and automorphism exhibit inter-iteration dependencies across the N loop, requiring accesses to some other or all N slots to calculate a new slot. Specifically, as in Figure 7 top, while the other operators can put the N loop at the outermost and pipeline on it, iNTT and NTT need all N elements for the $N \log N$ computations. Therefore forwarding and computing at each slot granularity are not feasible. The mismatched top-level loops between NTT/automorphism and other operators, called *orientation switches* in MAD [2], limit the scope of pipelining/sharing. If there is not enough SRAM buffer to transpose the intermediate polynomials on-chip, each orientation switch would need to spill the data to the off-chip memory. To avoid this cost, at least $O(N)$ buffer space, e.g., tens to hundreds of MB, is needed, even in MAD [2].

We leverage the widely known NTT decomposition method *in a novel way* to alleviate this issue. NTT decomposition, also known as four-step NTT, transforms the length-N data into a 2D shape, i.e., $N = N_1 \times N_2$. Then we sequentially perform N_1 instances of length-N_2 column (i)NTTs, $N_1 \times N_2$ element-wise twiddle multiplications, and N_2 instances of length-N_1 row (i)NTTs. Now the column and row (i)NTT steps have N_1 and N_2 independent instances, respectively, and can thus pipeline along these dimensions. Essentially, with our loop order notation, we transform the $\log N \triangleright N$ loop nest into $N_1 \triangleright \log N_2 \triangleright N_2 \rightarrow N_1 \triangleright N_2 \rightarrow N_2 \triangleright \log N_1 \triangleright N_1$.

Although in this sequence the outermost loops N_1 and N_2 do not match, the column and row (i)NTTs can each pipeline with their preceding and succeeding operators, respectively, as in Figure 7 bottom. For example, the middle part forms a row-iNTT $\rightarrow$ BConv $\rightarrow$ row-NTT pipeline, with a matched N_2 outermost loop. If we look at the data access patterns at the bottom of the figure, row-iNTT could stream in each of the $N_2 \times \alpha$ chunks one at a time, and perform the small length-N_1 row iNTT on it. BConv can access just α elements each time, for $N_1 \times N_2$ iterations. Both operators save the on-chip buffer capacity at least by a factor of N_2. The beginning and end parts also form two pipelines on the N_1 loop and save the buffer capacity by $N_1\times$. Orientation switches only happen two times in the middle of each decomposition, a $2\times$ saving compared to the top original case. The top spatial pipeline in Figure 6 also shows a real-case example. The values of N_1, N_2 are parameters decided by our scheduler (Section V-D).

C. Optimization: Hybrid Rotation

Recall that sharing auxiliary data requires multiple operators of the same type. The BSGS-based PtMatVecMult function seems to be a good candidate for such dataflow, since it contains many HRot operators with large evk constants, which incur significant latencies to load from the off-chip memory and dominate the bootstrapping performance. However, in the state-of-the-art execution schemes, these HRots *cannot*

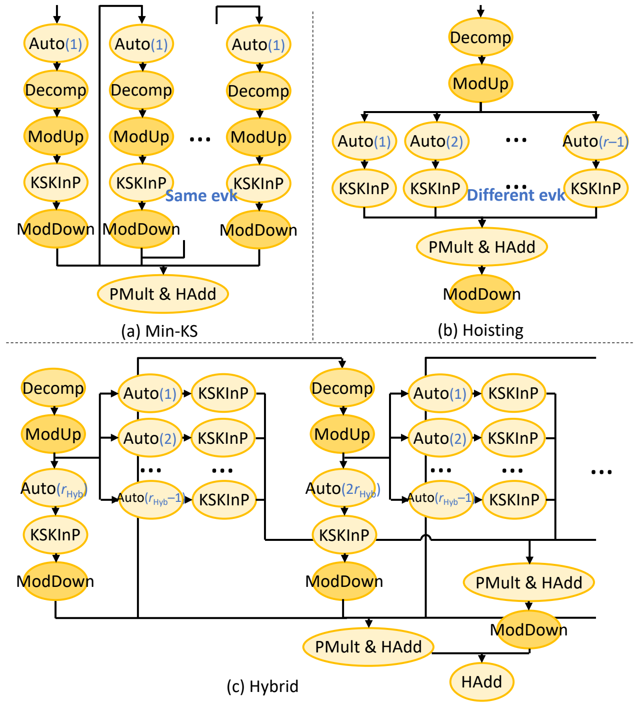


Fig. 8. Use hybrid rotation (c) to improve data sharing, which combines Min-KS by ARK [34] (a) and Hoisting by MAD [2] (b).

easily share the evk. ARK [34] proposed the *Min-KS* flow shown in Figure 8(a). To obtain all the baby-step HRots for $i = 1, 2, \ldots, n_1 - 1$ (Algorithm 1 Line 2), Min-KS sequentially rotates a unit amount in each step with the same evk, but these HRots have dependencies and can only execute one after another. Unless the full evk is cached with a large buffer space, we cannot reuse it with fine-grained sharing. MAD [2] proposed the *Hoisting* optimization shown in Figure 8(b). It shares (a.k.a., hoists) the common Decomp, ModUp, and ModDown operations among different HRots, but these parallel HRots have different rotation amounts and need different evks. Min-KS works better in large-SRAM scenarios, whereas Hoisting excels for small SRAM buffers [2].

We find that these two approaches can be combined into a hybrid rotation scheme to achieve the best of both worlds, as well as enabling evk sharing opportunities. As shown in Figure 8(c), we follow the Min-KS flow to perform coarse-step rotations, by distances of $r_{\text{Hyb}}, 2r_{\text{Hyb}}, \ldots$. Then, from each of these results we use the Hoisting scheme to do fine-step rotations by $1, 2, \ldots, r_{\text{Hyb}} - 1$. Here r_{Hyb} is a parameter to adjust the balance between the two schemes, determined by our scheduler (Section V-D). Note that this hybrid rotation has a similar pattern to the BSGS method, but is applied only to the baby steps, i.e., another decomposition within BSGS. Specifically, for n_1 baby steps in BSGS, there are $\lceil n_1/r_{\text{Hyb}} \rceil - 1$ coarse steps following Min-KS, and each coarse step has $r_{\text{Hyb}} - 1$ fine steps following Hoisting.

Compared to pure Min-KS, our hybrid rotation uses r_{Hyb} more evks, but saves $n_1 - \lceil n_1/r_{\text{Hyb}} \rceil$ ModUp and ModDown operations as the main benefits of using Hoisting. Compared to pure Hoisting, hybrid rotation does more ModUp and ModDown operations, but saves $n_1 - 1 - r_{\text{Hyb}}$ evks. In addition, now the fine-step HRots across all the coarse steps can share the same evks, providing more opportunities for our cross-operator dataflow optimizations.

D. Scheduling Algorithm

With the cross-operator dataflow design space discussed above, the CROPHE scheduler aims to identify the optimized scheme for the given FHE workload on the specific hardware. Following the three-level hierarchy of sequential execution $\rightarrow$ temporal pipelining/sharing $\rightarrow$ spatial pipelining/sharing, the scheduler builds all potential schemes in a bottom-up manner, and uses an analytical hardware cost model to estimate their performance to decide the best one. This scheduling approach is widely used in DNN accelerators [21], [29], [37], [42], [58]. We currently use simple exhaustive search, which can handle our workloads in reasonable time (e.g., searching ResNet-20 in 100 hours using *a single CPU core*). Scheduling is a one-time offline cost. We leave more efficient search to future work. Nevertheless, we already incorporate several techniques that help alleviate the long scheduling time issue in our current implementation, such as merging redundant computational subgraphs to search only once, as described below.

Bottom-up dataflow scheme composition. Given the computational graph of an FHE workload, we generate all sub-graphs up to a certain size (number of operators, e.g., 7 to 10 in our experiments), and enumerate all loop tiling schemes and orders for the operators in each sub-graph to compose the spatial pipelining/sharing groups at the bottom level. The number of possible loop orders is limited due to the few loops, so the total number of candidates is not too huge. For each spatial scheme, we pre-calculate its computation latency, PE register usage, and global SRAM buffer usage. We immediately discard the schemes whose data footprints exceed the hardware register file and/or global buffer capacity.

Then, we use the spatial groups to further compose temporal pipelining/sharing groups. We consider the spatial groups that share the same inputs or have adjacent dependencies, and also have some same top loops for fine-grained pipelining/sharing. Their global SRAM buffer usages are added up and compared to the hardware buffer capacity. Similarly, we compose the top-level sequentially executing groups.

With the above candidates for different sub-graphs of the workload, the final step is to concatenate them to cover the full graph. We use dynamic programming to extend the currently best dataflow scheme until a certain operator, with the next sub-graph starting from its successor.

For large FHE workloads like ResNet-20 and ResNet-110, the complete computational graph is too large to directly consider all possible sub-graphs. We thus pre-partition the graph into several smaller ones [41]. The above procedure is then conducted on each partitioned graph. In many workloads, there exist significant redundancies in the computational graphs; e.g., the same sub-graph of KeySwitch appears in many places. Pre-partitioning also merges these redundant cases and only searches once. The size limit of the partitioned graph represents a trade-off between search time and dataflow efficiency, and is set to 25 empirically in our implementation.

Support for novel optimizations. To support the NTT decomposition in Section V-B, the scheduler exploits different $N = N_1 \times N_2$ combinations when composing the spatial groups containing an NTT operator. Each NTT can use different decomposition. Different values of N_1, N_2 of an NTT would generate separate spatial groups. Fortunately, N_1 and N_2 should not be too small; otherwise the decomposed small NTTs cannot fully utilize the multiple lanes in the PE. Therefore the combinations are not many.

To support the hybrid rotation in Section V-C, the scheduler should exploit different r_{Hyb} values for each BSGS HRot. This requires significant graph-level transformations as in Figure 8. Thus we enumerate it at the very beginning and generate one computational graph for each r_{Hyb}. Then we conduct the above search on each graph separately.

Hardware cost model. Our scheduler optimizes the end-to-end execution time. For each spatial/temporal pipelining/sharing group, it carefully calculates its execution time with full consideration of both the computation and memory access latencies. The final time of a group is the maximum of the two. For computation, the fine-grained pipelining/sharing could partially overlap the execution of different operators. For memory accesses, we use the data amount and the memory bandwidth to derive the latency. Note that this simple analytical model is used only in the scheduler as an estimation. Our evaluation is done with detailed cycle-accurate simulation.

VI. Methodology

Implementation. We implement the CROPHE scheduler as described in Section V-D, which identifies the optimized cross-operator dataflow for an FHE workload on a specific hardware configuration. It is written in Python, with about 12890 lines of code. The scheduler outputs a dataflow result file that details the optimized spatial/temporal pipelining/sharing schemes for all the operators in the workload.

Then we use a mapper to map the dataflow scheme to the hardware accelerator. It follows the approaches described in Section IV-B for PE allocation and operator placement. It generates detailed trace files that describe the execution of each operator, which can then be fed to our simulator below.

For the CROPHE hardware accelerator, we implement its major logic units in RTL, and synthesize them using a 7 nm process [56]. We use FN-CACTI [47] to model the SRAM buffers and register files. The inter-PE NoC is modeled with Orion 3 [32]. All components are scaled to 7 nm according to the parameters from [48]. The PEs operate at 1.2 GHz in a fully pipelined manner. The SRAM banks are single-ported and run at doubled frequency of 2.4 GHz. The off-chip memory uses HBM stacks [30] to offer 1 TB/s total bandwidth, same as the baseline accelerators.

We also develop a cycle-accurate simulator for the CROPHE hardware accelerator for performance evaluation. The simulator takes the trace files generated by the mapper to drive

TABLE I
HARDWARE CONFIGURATIONS OF THE TWO CROPHE VARIANTS AND THE BASELINE ACCELERATORS.

	BTS [35]	**ARK [34]**	**CROPHE-64**	**CL+ [51]**	**SHARP [33]**	**CROPHE-36**
Word length (bits)	64	64	64	28	36	36
Logic frequency (GHz)	1.2	1	1.2	1	1	1.2
Number of lanes	-	256	256	512	256	256
Number of PEs (or clusters)	2048	4	64	8	4	128
DRAM bandwidth (TB/s)	1	1	1	1	1	1
SRAM bandwidth (TB/s)	38.4 + 292	20 + 72	39 + 314	84	36 + 36	44 + 354
SRAM capacity (MB)	512 + 22	512 + 76	512 + 16	256	180 + 18	180 + 8
Area w/o SRAM & PHY (mm^2)	89.5	116.6	129.4	-	89.4	105.5
Area (mm^2)	373.6	418.3	362.8	222.7	178.8	251.1
Power (W)	163.2	281.3	195.2	126.8	94.7	181.1

the execution. The operators on the PEs take pre-determined latencies that match the RTL implementation. We model the on-chip data transfers across the inter-PE NoC in a cycle-accurate way. Following the packet-based data communication scheme described in Section IV-A, the PEs transfer data packets according to the statically determined data dependencies between co-running operators. The off-chip memory access latencies are obtained from Ramulator 2 [40], using the specific memory access traces.

Configurations and baselines. We use two configurations of CROPHE to compare with the previous baseline FHE accelerators, as summarized in Table I. First, a 64-bit CROPHE is used to compare with BTS [35] and ARK [34], both of which also use 64-bit words. Second, a 36-bit CROPHE matches SHARP [33], and the same configuration is scaled to 28-bit (omitted in the table) to also compare with CraterLake [51]. Only CraterLake was at 14 nm so we scale it to 7 nm as reported in [33], denoted as CL+. We configure the two CROPHE variants to have similar area as the respective baselines. Table II shows the detailed breakdown of CROPHE-36. In Section VII-C we will reduce the global SRAM capacity to demonstrate CROPHE also supports scenarios with limited on-chip buffers. Note that from Table I it seems CROPHE has much more lanes $\times$ PEs than the baselines. This is because a CROPHE lane is very simple, only including a single modular multiplier and a modular adder/substractor, while a lane in the baselines may contain many dedicated functional units for NTT, BConv, automorphism, etc. So the total logic capabilities in CROPHE and baselines are still comparable.

For fair and comprehensive comparison, we carefully model and reproduce all the baselines on our own simulators. The reproduced results are slightly slower than those reported in the original papers. We believe this is due to our more realistic simulation of DRAM accesses in Ramulator. Furthermore, we consistently apply the state-of-the-art scheduling method, MAD [2], to all the baselines, further improving their performance. We also apply several known optimizations to all designs, such as on-the-fly limb extension (OF-Limb) for plaintexts [34] and pseudo-random number generation (PRNG) for evks [2], [51]. We use the same parameter set as each baseline originally uses when comparing with each, as listed in Table III. All of them are able to achieve 128-bit security.

TABLE II
AREA AND POWER BREAKDOWN OF CROPHE-36.

Component	**Area** (μm^2)	**Power** (mW)
256 modular multipliers	337650.31	388.80
256 modular adders/subtractors	27784.55	33.79
64 kB register files	67242.02	16.86
Inter-lane network	15806.76	58.17
PE	448483.64	497.62
	Area (mm^2)	**Power** (W)
128 PEs	57.40	63.70
Inter-PE NoC & crossbars	40.70	67.40
Global buffer	116.05	15.34
Transpose unit	7.38	2.87
HBM PHY	29.60	31.80
Total	251.13	181.11

TABLE III
PARAMETER SETS WHEN COMPARING WITH EACH BASELINE.

Parameter set	$\log_2 N$	L	L_{boot}	dnum	α
BTS (INS-2)	17	39	19	2	20
ARK	16	23	15	4	6
SHARP	16	35	27	3	12
CraterLake	16	59	51	1	60

Workloads. We evaluate four workloads, bootstrapping, HELR1024, ResNet-20, and ResNet-110, as the standard set of benchmarks used in the baselines. Bootstrapping measures the end-to-end bootstrapping time, which is a commonly used metric as many FHE workloads are dominated by the bootstrapping latency [33]. We use the optimized sparse-packed bootstrapping method [14]. HELR [24] trains a binary classification model using logistic regression. A 196-element weight vector is trained with a set of 14×14 MNIST images. For consistency with previous studies [33], we use 1024 images and train 32 iterations, and report the average execution time per iteration. ResNet-20 and ResNet-110 perform DNN inference on $32 \times 32 \times 3$ CIFAR-10 images [36] using the

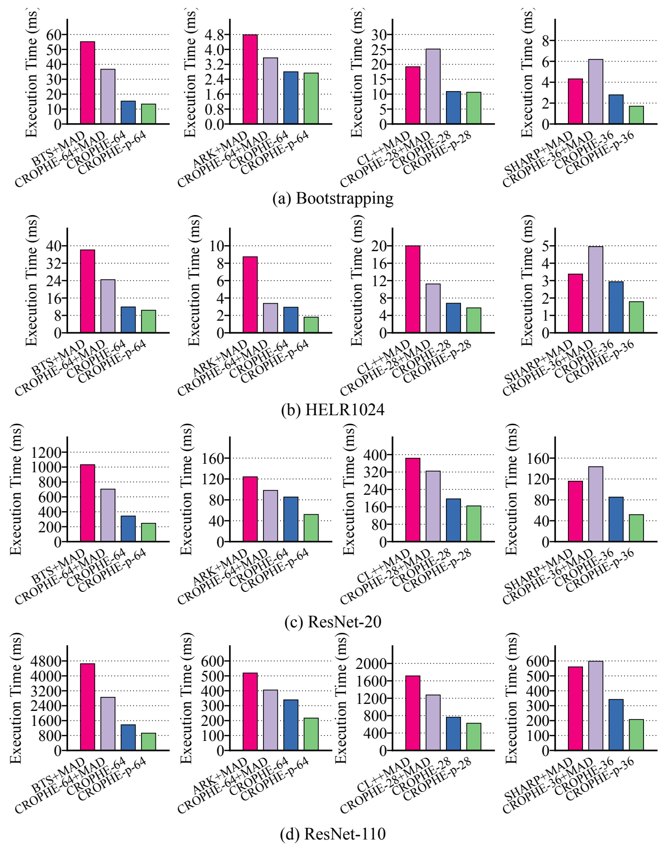


Fig. 9. Overall performance comparison between CROPHE and baselines.

CKKS implementation [38] of the corresponding ResNet models [26]. We specifically include ResNet-110 to demonstrate our scheduling techniques can scale to large-scale workloads.

VII. Evaluation

We now present the evaluation results of CROPHE when compared to prior accelerators and scheduling techniques. Besides the overall performance benefits (Section VII-A), we particularly highlight the improvements on hardware resource utilization (Section VII-B), the increasing contributions at smaller on-chip SRAM capacities (Section VII-C), and the effectiveness of individual hardware and scheduling optimizations proposed in this work (Section VII-D).

A. Overall Comparison

Figure 9 compares four designs: the baseline with MAD scheduling, our CROPHE hardware with MAD scheduling, CROPHE, and CROPHE-p. CROPHE-p statically partitions the PEs on the accelerator chip into a few clusters (e.g., 2 or 4, automatically determined by the scheduler), with all the clusters running the same workloads in a data-parallel fashion. Each cluster is scheduled as before. We find this partitioning helps better utilize the abundant hardware resources, and also allows more sharing for the auxiliary constant data like evks across clusters, as long as there exist independent input data.

TABLE IV
Resource utilization on different designs for ResNet-20.

Design	PEs	NoC b/w	SRAM b/w	DRAM b/w
ARK+MAD	39.48%	-	32.71%	68.54%
CROPHE-64	62.97%	64.45%	31.58%	61.32%
CROPHE-p-64	76.69%	72.72%	19.51%	70.62%
SHARP+MAD	41.73%	-	63.44%	51.83%
CROPHE-36	56.53%	61.64%	61.35%	43.20%
CROPHE-p-36	65.46%	74.06%	45.66%	49.88%

Overall, CROPHE exhibits significant performance improvements. On the four workloads, CROPHE-64 achieves 3.60×, 3.21×, 3.00×, and 3.38× speedups over BTS+MAD, and 1.71×, 2.97×, 1.45×, and 1.53× over ARK+MAD. Similarly, CROPHE-36 outperforms SHARP+MAD by 1.55×, 1.15×, 1.36×, and 1.64×. The data-parallel CROPHE-p is even faster, with 1.75×, 4.86×, 2.39×, and 2.39× speedups over ARK+MAD, and 2.54×, 1.89×, 2.24×, and 2.70× speedups over SHARP+MAD. These performance advantages are mainly due to the better cross-operator dataflow that significantly reduces data accesses to more expensive memory levels (off-chip DRAM, global SRAM) for both intermediate ciphertexts and auxiliary constants. More detailed data are in Figure 11 in Section VII-D.

Note that sometimes using our hardware architecture with MAD performs worse than the baselines, e.g., SHARP. This is because the MAD dataflow only has limited operator fusion and cannot create large operator pipelining/sharing groups to fully utilize the large amount of PEs in our hardware. In addition, when the intermediate data are not directly forwarded between PEs, they need to be written to the SRAM buffer or even the off-chip DRAM, and read back later. These problems substantially degrade performance. The CROPHE dataflow schemes are able to alleviate these issues, and thus necessary to unlock the full potential of our hardware architecture.

B. Resource Utilization

Table IV shows the utilization of PEs, NoC bandwidth, global SRAM buffer bandwidth, and off-chip DRAM bandwidth when executing ResNet-20 on different designs. When reproducing the baselines, for simplicity we assume idealized NoC performance, so this utilization is not shown. The PE utilization in CROPHE can reach 57% to 77%. The state-of-the-art baselines like SHARP can have 65% utilization for their NTT and element-wise engines, while the other dedicated units for BConv and automorphism have much lower utilization below 30% as reported in their original paper [33, Figure 6(b)]. This result demonstrates the benefit of higher resource utilization from our flexible homogeneous architecture.

For SRAM and DRAM bandwidth, the utilization remains similar to the baselines. This is because both the data access amount and the execution time have reduced. The former is because of the better cross-operator dataflow in CROPHE that better reuses data on-chip, as illustrated in Figure 11. The

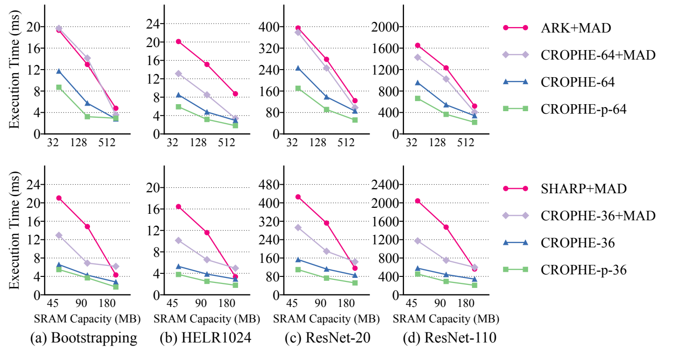


Fig. 10. Performance comparison between CROPHE and baselines at smaller SRAM capacities.

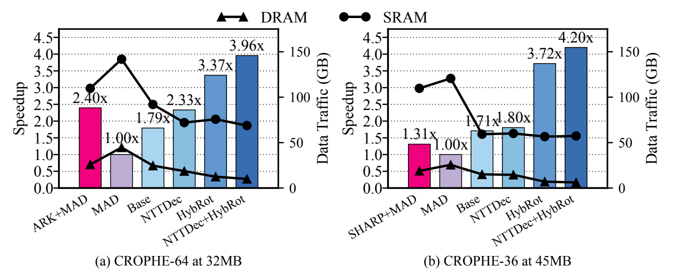


Fig. 11. Performance improvement breakdown of CROPHE techniques, and the corresponding data access traffic to SRAM and DRAM. Running the bootstrapping workload.

fewer accessed data can be fetched using the full bandwidth in shorter amounts of time.

Finally, CROPHE has more intensive on-chip data transfers due to more complex operator mapping. Therefore the NoC bandwidth exhibits high utilization and potentially bounds the performance. Fortunately, our heuristic mapping strategies in Section IV-B are able to deliver sufficiently good efficiency.

C. Performance at Smaller SRAM Capacities

The key benefit of CROPHE is to exploit the fine-grained data pipelining/sharing to reduce the required on-chip SRAM buffer space. Figure 10 reduces the global buffer capacities of the original 64-bit and 36-bit configurations, and compares with the best baseline in each case, i.e., ARK and SHARP. Generally, the speedups of CROPHE increase as the SRAM shrinks. Between CROPHE-36 and SHARP, the speedups at 45 MB increase to 3.20×, 3.11×, 2.78×, and 3.42× from 1.55×, 1.15×, 1.36×, and 1.64× at 180 MB.

Note that one surprising result is that in Figure 10(c) and (d), CROPHE-p-36 with 45 MB is even faster than SHAPR+MAD with 180 MB. There are several reasons. First, as explained before, the CROPHE techniques significantly improve the cross-operator data reuse and reduce the necessary SRAM capacity to buffer intermediate data. We notice that in SHARP+MAD, we only need about 90 MB to achieve near-minimal DRAM accesses after applying MAD (the MAD paper uses even more aggressive 32 MB designs [2]). The rest space is used mainly to keep the many HMult/HRot evks. CROPHE uses the hybrid rotation optimization to more efficiently handle these evks. This makes the non-data-parallel CROPHE-36 with 45 MB perform similarly to SHARP+MAD with 180 MB. Second, our data-parallel use of CROPHE further enables sharing the expensive evks across multiple independent tasks. This extra benefit makes CROPHE-p-36 even faster, eventually outperforming SHARP+MAD.

D. Performance Breakdown

Finally, we break down the impact of our various novel techniques for cross-operator dataflow scheduling in Figure 11, which runs the bootstrapping workload on the two CROPHE configurations at a small SRAM capacity. We first evaluate a design that only uses the CROPHE homogeneous hardware architecture, with the prior MAD dataflow (denoted as "MAD"). We use ARK's Min-KS [34] for HRot in this design. Performance drops by 2.40× and 1.31× compared to the ARK and SHARP baselines. These results indicate that, while the homogeneous architecture is simple to implement and potentially allows for higher utilization, without a specially optimized dataflow scheme, performance would still suffer since existing scheduling techniques fail to fully exploit its potential. Thus, *the homogeneous and unified hardware architecture and the comprehensive cross-operator dataflow optimizations are tightly coupled and should be jointly applied.* Second, by switching from MAD to our basic cross-operator dataflow in Section V-A (denoted as "Base"), we can roughly match the baselines, with 0.75× to 1.31× speedups. These are because of the significant decreases in the SRAM and DRAM accesses as shown in the figure. These results demonstrate that even the basic cross-operator dataflow framework in CROPHE is more comprehensive and optimized than the baseline MAD. Further enabling the NTT decomposition (Section V-B, denoted as "NTTDec") and hybrid rotation (Section V-C, denoted as "HybRot") optimizations leads to further speedups and data access reduction. Between the two, the benefits of hybrid rotation are more significant than those of NTT decomposition, as managing evks is highly critical with limited SRAM space. In this scenario, CROPHE adaptively uses a hybrid rotation scheme close to Hoisting to reduce computations, as the evk cannot fit on-chip even with the Min-KS technique. Finally, combining the two can result in the best performance, which is 1.65× and 3.21× faster than the corresponding baselines.

VIII. Conclusions

We explore the opportunities of cross-operator dataflow optimization for FHE accelerators in this paper, and present the CROPHE framework. CROPHE discovers efficient cross-operator dataflow schemes with spatial/temporal data pipelining/sharing among FHE operators, and maps them onto a homogeneous and unified hardware architecture. Two novel optimizations for NTT and HRot are also proposed to further improve the efficiency. CROPHE is able to achieve better performance with reduced data movement cost compared to state-of-the-art accelerators and dataflow schemes.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author.

References

[1] R. Agrawal, A. Chandrakasan, and A. Joshi, "HEAP: A Fully Homomorphic Encryption Accelerator with Parallelized Bootstrapping," in *51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 756–769.

[2] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption," in *56th Annual International Symposium on Microarchitecture (MICRO)*, 2023, p. 685–697.

[3] 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 *29th International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 882–895.

[4] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941–956, 2021.

[5] 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*, 2017, pp. 423–442.

[6] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Advances in Cryptology – CRYPTO 1986*, 1987, p. 311–323.

[7] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys," in *Advances in Cryptology – EUROCRYPT 2021*, 2021, pp. 587–617.

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *3rd Innovations in Theoretical Computer Science Conference (ITCS)*, 2012, p. 309–325.

[9] Z. Brakerski and V. Vaikuntanathan, "Efficient Fully Homomorphic Encryption from (Standard) LWE," Cryptology ePrint Archive, Paper 2011/344, 2011. [Online]. Available: https://eprint.iacr.org/2011/344

[10] J. Cai, Y. Wei, Z. Wu, S. Peng, and K. Ma, "Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators," in *50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[11] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, vol. 62, no. 1, pp. 157–166, 2015.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[13] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[14] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for Approximate Homomorphic Encryption," in *Advances in Cryptology – EUROCRYPT 2018*, 2018, pp. 360–384.

[15] 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*, 2019, pp. 347–368.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017*, 2017, pp. 409–437.

[17] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing," in *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019, p. 142–156.

[18] X. Deng, S. Fan, Z. Hu, Z. Tian, Z. Yang, J. Yu, D. Cao, D. Meng, R. Hou, M. Li, Q. Lou, and M. Zhang, "Trinity: A General Purpose FHE Accelerator," in *57th International Symposium on Microarchitecture (MICRO)*, 2024, pp. 338–351.

[19] J. Dong, X. Chen, and M. Gao, "A Unified Vector Processing Unit for Fully Homomorphic Encryption," in *Design, Automation & Test in Europe Conference (DATE)*, 2025.

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

[21] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators," in *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, p. 807–820.

[22] K. Garimella, Z. Ghodsi, N. K. Jha, S. Garg, and B. Reagen, "Characterizing and Optimizing End-to-End Systems for Private Inference," in *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 3*, 2023, p. 89–104.

[23] S. Halevi and V. Shoup, "Faster Homomorphic Linear Transformations in HElib," in *Advances in Cryptology – CRYPTO 2018*, 2018, pp. 93–120.

[24] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic Regression on Homomorphic Encrypted Data at Scale," in *31st Innovative Applications of Artificial Intelligence Conference (IAAI)*, 2019, pp. 9466–9471.

[25] K. Han and D. Ki, "Better Bootstrapping for Approximate Homomorphic Encryption," in *Topics in Cryptology – CT-RSA 2020*, 2020, pp. 364–390.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[27] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search," in *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 943–958.

[28] W. Hu, Z. Lu, A. Jantsch, and H. Liu, "Power-Efficient Tree-Based Multicast Support for Networks-on-Chip," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011, pp. 363–368.

[29] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, "CoSA: Scheduling by Constrained Optimization for Spatial Accelerators," in *48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.

[30] JEDEC Solid State Technology Association, "High Bandwidth Memory DRAM (HBM3) (JESD238)," JEDEC Standard, 2022.

[31] N. E. Jerger, L.-S. Peh, and M. Lipasti, "Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support," in *35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, p. 229–240.

[32] A. B. Kahng, B. Lin, and S. Nath, "ORION3.0: A Comprehensive NoC Router Estimation Tool," *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 41–45, 2015.

[33] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption," in *50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[34] 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 *55th International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.

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

[36] A. Krizhevsky and G. Hinton, "Learning Multiple Layers of Features from Tiny Images," Technical Report, University of Toronto, 2009. [Online]. Available: https://www.cs.utoronto.ca/~kriz/learning-features-2009-TR.pdf

[37] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach," in *52nd International Symposium on Microarchitecture (MICRO)*, 2019, p. 754–768.

[38] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-Complexity Deep Convolutional Neural Networks on Fully Ho-

momorphic Encryption Using Multiplexed Parallel Convolutions," in *39th International Conference on Machine Learning (ICML)*, 2022, pp. 12 403–12 422.

[39] L. Lu, N. Guan, Y. Wang, L. Jia, Z. Luo, J. Yin, J. Cong, and Y. Liang, "TENET: A Framework for Modeling Tensor Dataflow Based on Relation-Centric Notation," in *48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 720–733.

[40] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu, "Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 112–116, 2024.

[41] O. Moreira, M. Popp, and C. Schulz, "Graph Partitioning with Acyclicity Constraints," *arXiv preprint arXiv:1704.00705*, 2017. [Online]. Available: https://arxiv.org/abs/1704.00705

[42] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *2019 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[43] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, p. 252–264, 1968.

[44] Prasetiyo, A. Putra, and J.-Y. Kim, "Morphling: A Throughput-Maximized TFHE-based Accelerator using Transform-Domain Reuse," in *30th International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 249–262.

[45] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An End-to-End Streaming Architecture with Two-Level Ciphertext Batching for Fully Homomorphic Encryption with Programmable Bootstrapping," in *56th Annual International Symposium on Microarchitecture (MICRO)*, 2023, p. 1319–1331.

[46] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 519–530.

[47] D. P. Ravipati, R. Kedia, V. M. Van Santen, J. Henkel, P. R. Panda, and H. Amrouch, "FN-CACTI: Advanced CACTI for FinFET and NC-FinFET Technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 3, pp. 339–352, 2022.

[48] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference," in *27th International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 26–39.

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

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

[51] 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 *49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, p. 173–187.

[52] N. Samardzic and D. Sanchez, "BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators," in *29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, 2024, p. 137–150.

[53] F. A. Samman, T. Hollstein, and M. Glesner, "Adaptive and Deadlock-Free Tree-Based Multicast Routing for Networks-on-Chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 7, pp. 1067–1080, 2010.

[54] 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 *25th International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 387–398.

[55] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 541–552.

[56] V. Vashishtha, M. Vangala, and L. T. Clark, "ASAP7 Predictive Design Kit Development and Cell Design Technology Co-optimization: Invited Paper," in *36th International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 992–998.

[57] C. Wang and M. Gao, "SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition," in *42nd International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[58] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, p. 369–383.

[59] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical Homomorphic Encryption Accelerator," in *29th International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 870–881.

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

[61] M. Zhou, Y. Nam, X. Wang, Y. Lee, C. Wilkerson, R. Kumar, S. Taneja, S. Mathew, R. Cammarota, and T. Rosing, "UFC: A Unified Accelerator for Fully Homomorphic Encryption," in *57th International Symposium on Microarchitecture (MICRO)*, 2024, pp. 352–365.