2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 979-8-3503-7638-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISPASS61541.2024.00016

CiFlow: Dataflow Analysis and Optimization of Key Switching for Homomorphic Encryption

Negar Neda*, Austin Ebel*, Benedict Reynwar†, and Brandon Reagen*
*New York University, New York, New York, USA, {negar, abe5240, bjr5}@nyu.edu
†Information Sciences Institute, University of Southern California, Arlington, VA, USA, breynwar@isi.edu

***Abstract*—Homomorphic encryption (HE) is a privacy-preserving computation technique that enables computation on encrypted data. Today, the potential of HE remains largely unrealized as it is impractically slow, preventing it from being used in real applications. A major computational bottleneck in HE is the key-switching operation, accounting for approximately 70% of the overall HE execution time and involving a large amount of data for inputs, intermediates, and keys. Prior research has focused on hardware accelerators to improve HE performance, typically featuring large on-chip SRAMs and high off-chip bandwidth to deal with large scale data.**

In this paper, we present a novel approach to improve key-switching performance by rigorously analyzing its dataflow. Our primary goal is to optimize data reuse with limited on-chip memory to minimize off-chip data movement. We introduce three distinct dataflows: Max-Parallel (MP), Digit-Centric (DC), and Output-Centric (OC), each with unique scheduling approaches for key-switching computations. Through our analysis, we show how our proposed Output-Centric technique can effectively reuse data by significantly lowering the intermediate key-switching working set and alleviating the need for massive off-chip bandwidth. We thoroughly evaluate the three dataflows using the RPU, a recently published vector processor tailored for ring processing algorithms, which includes HE. This evaluation considers sweeps of bandwidth and computational throughput, and whether keys are buffered on-chip or streamed. With OC, we demonstrate up to 4.16× speedup over the MP dataflow and show how OC can save 12.25× on-chip SRAM by streaming keys for minimal performance penalty.

I. Introduction

Today, many computations are outsourced to the cloud, whether to leverage its scale or overcome performance constraints of locally available devices (e.g., smartphones). Despite the success of this computational model, it remains vulnerable to attack and does not provide users with strong privacy and security guarantees over who can view and use their data. Homomorphic encryption (HE) [1] offers a solution. With HE, functions computed on ciphertext are also directly applied to the underlying plaintext. This facilitates direct computation on encrypted data, extending cryptographic security from communication and storage to include computation for complete end-to-end secure outsourced computation.

Many HE schemes (e.g., BGV [2], BFV [3], [4], and CKKS [5]) and software implementations (e.g., OpenFHE [6], Lattigo [7], SEAL [8]) now exist and have made steady performance improvements over the years. However, practical deployment of HE is still limited by significant performance overheads, attributed to large data sizes, complex operations, and additional functions to process. When encrypting and processing HE ciphertexts, the data becomes significantly larger than plaintext, forming large ciphertext vectors of up to 2^{17} elements. Here, each plaintext element is represented by hundreds to thousands of bits, and special precomputed keys ($\mathbf{evks}$) are needed for certain operations (i.e., multiplication and rotation) that can be upwards of hundreds of MBs. This puts substantial pressure on the memory system. Next, HE requires complex modular arithmetic for processing each element, which is not natively supported by mainstream commercial processors. Finally, HE operations are not one-to-one conversions of plaintext operators; they involve multiple additional functions for performance (e.g., Number Theoretic Transforms (NTT)) and correctness before and after the actual operation (e.g., multiplication and rotation). These factors collectively overwhelm commodity hardware, and prior work has consistently reported 4-6 orders of magnitude slowdown compared to plaintext [9]–[12].

The slowdown has been addressed by a growing body of work on hardware acceleration for HE. Many solutions now exist that consider fixed-function pipelines [9], [13], vector architectures [10], [14], [15], and tiled architectures [16], [17]. The architectural approaches taken differ, but the building blocks are the same: very large on-chip memories and high off-chip bandwidth. For example, CraterLake [14]) uses 256 MB of on-chip SRAM and assumes two HBM2e PHYs for a total bandwidth of 1TB/s. BTS [16] and ARK [17] both have 512 MB of on-chip SRAM and assume 1TB/s of memory bandwidth. The large on-chip SRAMs and multiple PHYs result in large chips, adding the cost of the advanced memory technology results in expensive solutions.

Our hypothesis is that by optimizing HE dataflow, we can capture on-chip reuse with far less SRAM and simultaneously reduce the off-chip bandwidth requirements. To understand the potential of optimizing dataflow, we deeply analyze the *hybrid key-switching algorithm (HKS)* [18]. HKS is the core computation of HE and highly complex. The use of HKS in HE is widespread; it is called after each rotation and homomorphic multiplication and is heavily used in bootstrapping. For example, recent work has reported that a single HE ResNet-20 inference takes 3,306 rotations [19], and prior research has further shown that HKS dominates HE's runtime [17], [20], [21], and can be up to 70% for private neural inference [19].

A single HKS execution can involve hundreds of NTTs, hundreds of MBs of input and output data, nearly 500MB of constant $\mathbf{evks}$, and up to 1GB of intermediate data. Thus, existing solutions to processing the workload effectively rely on large on-chip SRAMs and high off-chip bandwidth.

To understand and optimize HKS, we propose three distinct dataflows: Max-Parallel (MP), Digit-Centric (DC), and Output-Centric (OC). Our key insight is that with the OC dataflow, the intermediate state of HKS can be significantly compressed while maintaining high parallelism to utilize computational units. To demonstrate OC's potential, we implement five parameterizations of HKS taken from recent work following each of the three presented dataflows on a recent vector HE accelerator [15]. Using previously validated simulation infrastructure, we rigorously evaluate the dataflows. First, we assume a large on-chip SRAM (i.e., 392 MB), which is sufficient to buffer all $\mathbf{evks}$ on-chip while reserving 32MB for data (i.e., inputs and intermediates). We find that OC consistently outperforms other dataflows and can deliver up to 4.16$\times$ speedup over MP using the same bandwidth. Next, given that $\mathbf{evks}$ have no reuse within HKS, we elect to stream them on-chip and reserve a fraction of off-chip bandwidth for them. Here again, OC performs well. With our OC dataflow, we can stream $\mathbf{evks}$ to reduce on-chip SRAM by 12.25$\times$ and bandwidth by 3.3$\times$, while achieving the same performance as an MP on-chip implementation.

In summary, our contributions are:

1) We propose three different dataflows for HKS algorithm, named Max-Parallel, Digit-Centric, and Output-Centric. Through our analysis we demonstrate how strategically scheduling instructions and efficiently reusing loaded and generated data by the OC dataflow can lead to substantial off-chip bandwidth saving.
2) We evaluate two scenarios for handling $\mathbf{evks}$: streaming from off-chip and on-chip storage with a larger memory. Streaming $\mathbf{evks}$ reduces SRAM area by 12.25$\times$ and OC saves 3.3$\times$ bandwidth over the MP baseline.
3) We evaluate the performance of dataflows across different off-chip bandwidths and computational throughput to understand the bandwidth-compute trade-off. We find OC to be highly effective, matching naive MP HKS implementation with significant bandwidth saving. Furthermore, increasing accelerator throughput enhances performance even further.

II. Background

In this section we briefly introduce the CKKS HE scheme [5], including its key parameters and operations, using terminology from ARK [17] and Castro [20] when applicable. Unlike prior HE schemes, BGV [2] and BFV [3], [4], CKKS operates on vectors of real or complex numbers, rather than integers. To perform these operations, a vector message, $\mathbf{m}$, is first encoded into a plaintext *polynomial* $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$, represented as $[\mathtt{P}]$. In essence, this is another vector of length N, with each element being an integer no larger than Q. Depending on a problem's complexity, the value of N typically ranges from 2^{11} to 2^{17}, while Q can range from hundreds to thousands of bits. For a polynomial with N coefficients, a vector message must be of length $n \leq {}^{N}/_{2}$.

Table. I: Relevant CKKS Parameters.

Param.	Description
$\mathbf{m}$	Message, a vector of real or complex numbers.
$[\![\mathbf{m}]\!]$, $[\![\mathbf{m}]\!]_s$	Ciphertext encrypting the message, $\mathbf{m}$, under secret key s.
N	Power-of-two polynomial ring degree.
n	Length of the vector message, $n \leq {}^{N}/_{2}$.
Q	Initial polynomial modulus.
P	Auxiliary modulus used in key-switching.
L	Maximum multiplicative level of $[\![\mathbf{m}]\!]$.
ℓ	Current multiplicative level and remaining *towers*.
K	Number of moduli/towers in P.
q_i	Small moduli in RNS decomposition of $Q = \prod_{i=0}^{L} q_i$.
p_i	Small moduli in RNS decomposition of $P = \prod_{i=0}^{K-1} p_i$.
$\mathtt{P}$	Polynomial in $\mathbb{Z}[X]/(X^N + 1)$.
$\mathcal{B}_\ell$, $[\mathtt{P}]_{\mathcal{B}_\ell}$	The set of primes $\{q_0, q_1, \ldots, q_\ell\}$, $[\mathtt{P}]_{i \in \mathcal{B}_\ell}$.
$\mathcal{C}$, $[\mathtt{P}]_{\mathcal{C}}$	The set of primes $\{p_0, p_1, \ldots, p_{K-1}\}$, $[\mathtt{P}]_{i \in \mathcal{C}}$.
$\mathcal{D}_\ell$, $[\mathtt{P}]_{\mathcal{D}_\ell}$	The union of $\mathcal{B}_\ell \cup \mathcal{C}$, $[\mathtt{P}]_{i \in \mathcal{D}_\ell}$.
$dnum$	Number of *digits* that $\mathtt{P}$ is decomposed into.
α	The number of towers in each digit, $\lceil {}^{(L+1)}/_{dnum} \rceil$.
$\mathbf{evk}$	Evaluation key used to convert $[\![\mathbf{m}]\!]_{s'}$ to $[\![\mathbf{m}]\!]_s$.

A ciphertext, $[\![\mathbf{m}]\!]$, consists of a pair of polynomials, (C_0, C_1), where one polynomial contains the message with a small amount of random noise added to it to ensure the security of the RWLE scheme. For efficiency, ciphertexts are decomposed into an equivalent RNS representation [22] consisting of many smaller, machine-word size moduli, q_i, such that $R_{q_0} \times R_{q_1} \times \ldots \times R_{q_\ell} = R_Q$. Each small moduli typically ranges from 36 to 64 bits [23], with larger moduli being more robust to accumulated noise. We can think of the RNS representation as an $N \times \ell$ matrix, with ℓ being the number of *levels* or *towers* in the current ciphertext.

CKKS supports several arithmetic operations like addition and multiplications between plaintext polynomials or ciphertexts. Additions and multiplications are simple pointwise operations between coefficients and therefore require the same number of levels in both operands. CKKS also supports *rotations* that cyclically rotate elements within the vector message by a specified amount, r. Notably, indexing vector elements is not efficiently supported and therefore ciphertext rotations are the primary way of computing fully connected layers and convolutions in neural networks.

Rotations and multiplications transform the ciphertext so that it cannot be decrypted by the original secret key, s. As a result, an auxiliary process known as "key-switching" is required, which is described in detail in the following section. This operation involves many NTTs, which are analogous to FFTs. A naive key-switching implementation can have poor operational intensity and bottleneck many practical applications. For example, recent work shows nearly 70% of execution time is spent performing key-switching operations for ResNet-20 [19].

III. Hybrid Key-Switching

In this section, we describe the *Hybrid* key-switching (HKS) algorithm, a crucial step after performing ciphertext rotations and ciphertext-ciphertext multiplications. Both operations convert a ciphertext encrypted by a secret key, s, into a new ciphertext only decryptable by a new secret key, s'. To continue computation, the ciphertext must be returned to a form decryptable by the original key, s. This process, known as key-switching, involves multiplying the ciphertext $[\![\mathbf{m}]\!]_{s'}$ with a special evaluation key, $\mathbf{evk}$, which re-encrypts it from s' to s. However, this process introduces significant noise growth, which can be managed using techniques proposed by Fan [4] and later by Han [18]. These techniques involve performing the $\mathbf{evk}$ multiplication at a higher modulus, reducing noise growth but increasing HE computational complexity with complex NTTs and RNS basis extensions.

HKS [18] is a generalization of existing key-switching techniques that let the user trade off the complexity of key-switching with the size of the $\mathbf{evk}$. It has been widely adopted both in software implementations [24], [25] and in recent HE accelerators [14], [16], [17], [21] yet the aforementioned trade-off, especially in hardware, is still not well understood.

A. Overview

Broadly, HKS is composed of two phases that we denote as *ModUp* and *ModDown*. The *ModUp* phase can be broken down into five consecutive stages, P1-P5, and the *ModDown* phase into four, P1-P4. We use terms such as *ModUp/Down_Pi* to refer to the *i'th* stage of the *ModUp/Down* phase. These stages can be seen in Figure 1. We loosely adopt this terminology from Han [18] to intuitively represent that the first phase of HKS is geared towards extending the RNS base from Q_ℓ to PQ_ℓ, whereas the second phase is focused on reducing the modulus back from PQ_ℓ to Q_ℓ.

We consider an input polynomial as a matrix of size $(N \times \ell)$, which changes shape throughout the key-switching process, depending on the chosen parameters. Figure 1 represents the HKS dataflow for a specific parameter set; the core stages of HKS remain the same across different parameter choices. In Section V-B, we explore a wider range of parameter sets and note that the insights we gain from analyzing this specific case remain applicable. We highlight a single tower of the input polynomial in red in Figure 1, with the widths of subsequent stages accurately reflecting the change in sizes of the original $(N \times \ell)$ input polynomial. For example, a preliminary step in HKS decomposes the input polynomial into $dnum$ digits, each of size roughly, $\alpha = \lceil (L+1)/dnum \rceil$, reshaping the input matrix to $(N \times \ell)$. In Figure 1, for which $dnum = 3$, the input polynomial is decomposed into *three* different colored digits, with each digit being 11 towers wide.

B. ModUp

$(\mathbf{P1})$ *INTT*: Following the initial digit decomposition step, each tower must undergo an *INTT*, which converts the polynomial from the *evaluation* domain to the *coefficient* domain. This conversion has a complexity of $\mathcal{O}(N \log N)$, similar to the FFT, and is applied separately to all ℓ towers.

$(\mathbf{P2})$ *BConv*: Each digit, now in the coefficient domain, goes through a basis conversion to change the set of primes representing a polynomial. The number of towers is extended from α to β, where $\beta = \ell + K - \alpha$, and K is the number of towers in moduli P. The number of modular multiplications in this stage is roughly $N \times \alpha \times \beta$ for each of the $dnum$ digits.

$(\mathbf{P3})$ *NTT*: After basis extension, the digits once again return to the evaluation domain through the *NTT*. The computational complexity, similar to the *INTT* stage, is $\mathcal{O}(N \log N)$, performed independently for $\beta \times dnum$ towers. This output is concatenated with the original digit to form a new polynomial modulo PQ_ℓ. Our new matrix is now of size $dnum \times N \times (\ell + K)$.

$(\mathbf{P4})$ *Apply Key*: Now that the polynomial is in a larger modulus, we can multiply it with $\mathbf{evk}$ with negligible added error. This is a point-wise operation with the $\mathbf{evk}$ being of shape $dnum \times 2 \times N \times (\ell + K)$. In practice, key sizes typically range from 100MB to 400MB (see Table III).

$(\mathbf{P5})$ *Reduce*: The last stage in the *ModUp* phase sums each digit's output from $P4$ into one final matrix of size $2N(\ell + K)$.

C. ModDown

We now have two polynomials, each of size $N \times (\ell + K)$ that must be reduced to their original size of $N \times \ell$ for further computation. This process begins by taking the last K towers of each polynomial and performing a similar series of *INTT* $\rightarrow$ *BConv* $\rightarrow NTT$ operations. Here, each *BConv* operation converts the number of towers from K to ℓ. The runtime complexity of these three stages is $2K \times N \log N$, $2N \times K \times \ell$, and $2\ell \times N \log N$, respectively. These polynomials go through one final scalar-tower multiplication and summation to finish out the key-switching process.

IV. CiFlow: A Taxonomy and Analysis of HKS Dataflow

We propose three dataflows for the HKS algorithm: Max-Parallel (MP), Digit-Centric (DC), and Output-Centric (OC). These dataflows differ in their sequence of instructions, reuse of loaded and computed data, intermediate data generation, and off-chip memory interaction. Assuming unlimited on-chip memory, the performance gap between these dataflows would decrease significantly. This is because all inputs and intermediate data could be stored on-chip, making MP even more advantageous than DC and OC due to its highly parallel nature. (E.g., this is the dataflow Cheetah [9] used to demonstrate the FHE overhead could be overcome with large chips.) Our goal is to show that by changing the sequence of operations we can have the same performance using a smaller on-chip memory and less off-chip bandwidth. In this section, we provide a detailed explanation of the principles and strategies behind each dataflow. Later, we evaluate their performance by varying bandwidth and computational throughput. Our analysis highlights how dataflow optimizations help to save memory and bandwidth resources.

A. Max-Parallel Dataflow (MP)

This dataflow is designed to prioritize kernel parallelism at all costs. Each tower is loaded onto the on-chip memory

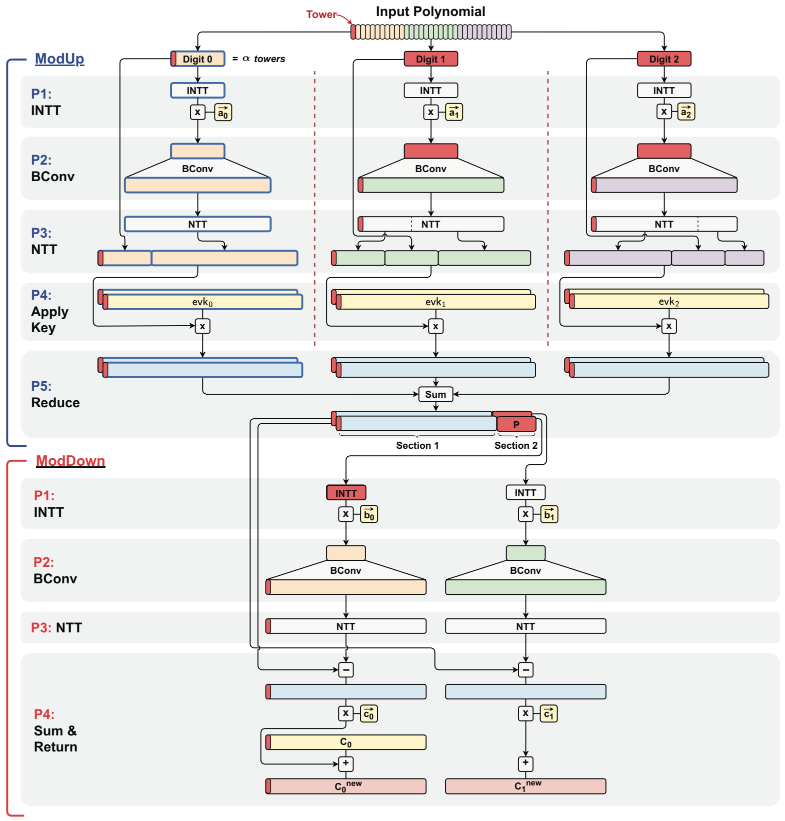

Figure. 1: Hybrid key-switching dataflow diagram for parameters, $\ell = 33,\ \alpha = 11,\ dnum = 3$.

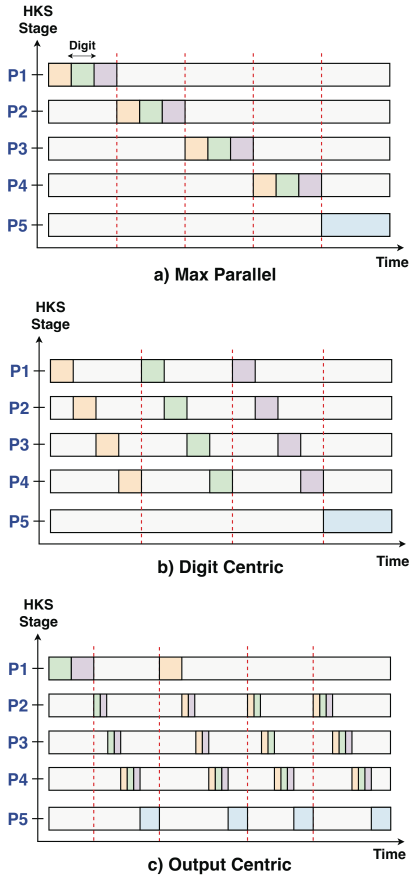

Figure. 2: High-level *ModUp* timing diagrams for the three proposed dataflows.

for a single operation, and each operation is executed on all input towers sequentially before processing the next step. However, this approach comes with a drawback. Specifically, in *ModUp_P2* and *ModDown_P2*, the generated intermediate data after BConv becomes extremely large, even though only one of the BConv output towers is required for calculating each output tower. For BTS3, at least 675MB of on-chip memory is required to prevent excessive load and stores to the off-chip. MP was used by prior work (e.g., Cheetah [9], and HEAX [13]), and we use it as the baseline HKS implementation.

B. Digit-Centric Dataflow (DC)

This dataflow adopts a "one-digit-at-a-time" approach, where each digit is loaded onto the chip and all possible calculations involving that digit are performed before moving on to the next digit. As illustrated in Figure 2(b), all P1 to P5 calculations for a single digit are done in sequence, maximizing data reuse. The blue frames in Figure 1 show the intermediate data generated from processing a single digit in the *ModUp* section, which is used for calculating a partial product of the *ModUp_P5* step. As depicted in Figure 1, BConv still expands the digit from α to β towers, generating a partial product of the output that can either be stored on-chip for later reduction to save bandwidth or sent off-chip to minimize on-chip memory requirements. Once *ModUp* is completed, *ModDown* follows the same approach. For BTS3, which is the largest benchmark, DC requires 255MB of on-chip memory, which is 62% less compared to MP dataflow.

C. Output-Centric Dataflow (OC)

Finally, we propose Output-Centric (OC) dataflow, which is optimized for data movement and on-chip storage, leveraging insights from the first two solutions. OC focuses on computing one output tower at a time, efficiently utilizing on-chip resources. In this dataflow, the *ModUp* stage has two sections: Section1 computes *ModUp* output towers in modulo Q, with $dnum - 1$ digits going through *ModUp_P2 & ModUp_P3* step and one tower from the last digit being bypassed through *ModUp_P2 & ModUp_P3*. As shown by the red towers in Figure 1, to generate the first output tower, the first digit is bypassed, while the other two digits must pass through the *ModUp_P2* stage. Section2 computes *ModUp* output towers in mod P, requiring all digits to pass through *ModUp_P2 & ModUp_P3* for a single output calculation.

In both sections, the computation is optimized to minimize on-chip memory requirements. Since only one output tower is calculated at a time, the entire computation of *ModUp_P2*

Table. II: DRAM transfers (MB), including $\mathbf{evk}$ with 32MB on-chip memory and Arithmetic Intensity (AI) in ops/byte.

Benchmark	MP		DC		OC	
	MB	AI	MB	AI	MB	AI
BTS1	600	1.81	600	1.81	420	2.59
BTS2	1352	1.14	1278	1.2	716	2.15
BTS3	1850	1.00	1766	1.04	1119	1.65
ARK	432	1.05	356	1.27	180	2.52
DPRIVE	365	1.26	336	1.37	170	2.71

& ModUp_P4 is unnecessary. The red towers in Figure 1 represent the computations needed to generate one output tower, with *ModUp* being in Section1. As shown, there is no need to do all calculations of *ModUp_P2 & ModUp_P4*, minimizing the memory requirement and off-chip data communication. Additionally, for *ModUp_P5*, only one partial tower is calculated at a time, allowing them to be on-chip for accumulation and only store back the accumulation result.

In Section2, where all digits are required for calculating a single output tower in *ModUp*, we have used the following strategy to manage the on-chip memory limitation. Since the INTT of the first $dnum - 1$ digits are already on-chip, we compute the partial sum with those digits and then the final digit is loaded to compute the last partial sum and the final output towers. This approach reduces off-chip memory interaction and on-chip memory requirement.

ModDown has the same approach as *ModUp*, loading all towers related to $[\mathtt{P}]_{\mathcal{C}}$ on-chip. Calculating one output tower at a time eliminates the expansion of *ModDown_P2*, enhancing the efficiency of HKS with a small on-chip memory.

D. Dataflow Comparison: Arithmetic Intensity

Table II evaluates the off-chip data movement, including $\mathbf{evks}$ and input/output data, of each benchmark. Here, the assumed on-chip memory is 32MB with $\mathbf{evks}$ being streamed on-chip. Across all benchmarks, we see that OC can significantly reduce the total off-chip traffic. The number of operations per HKS benchmark is independent of dataflow. Thus, we can see that the arithmetic intensity (AI) of each is also significantly improved with OC.

In a recent study, MAD [26] introduced techniques to optimize and accelerate FHE. Analyzing their optimizations, we find their solution is analogous to our proposed DC dataflow. According to the reported arithmetic intensity in Table II, leveraging the OC dataflow results in 1.43× to 2.4× more arithmetic intensity than MP and 1.43× to 1.98× more than DC. Furthermore, MAD [26] (and other FHE accelerators) also consider a technique for key compression, which halves the off-chip data movement of keys. Incorporating key compression in our approach will further boost our AI to 3.82.

V. Methodology

This section provides an overview of our dataflow implementations. We will start by describing the hardware architecture and then explain the benchmarks used for performance evaluation. Finally, we will introduce the software framework employed for generating HKS instructions for different benchmarks and dataflows.

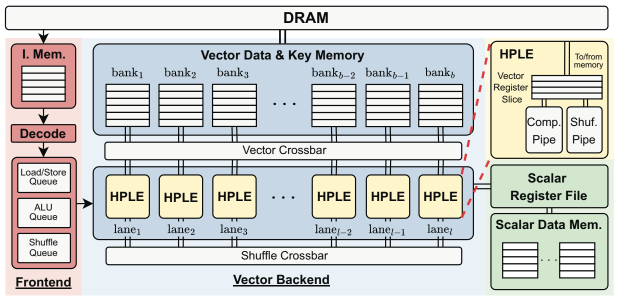

Figure. 3: Microarchitecture of the RPU.

A. Hardware Simulation

We implement the dataflows introduced in Section IV on the RPU [15] and conduct an extensive evaluation in Section VI. The RPU is a vector processor supporting Ring-Learning-With-Errors (RLWE)-based algorithms, including HE, with 64, 64-bit vector registers, 64, 64-bit scalar registers, a 32MB vector data memory, and a 1MB scalar data memory, operating at 1.7GHz. These exist alongside an additional modulus register file, which stores a set of RNS moduli. Figure 3 presents the RPU microarchitecture. It consists of a front-end to handle instruction fetching, decoding, and control logic, and a backend that contains the high-performance large arithmetic word engines (HPLEs) to efficiently perform HE operations. A parameterized simulator is used to test different RPU configurations (e.g., SRAM capacity and computational throughput), and supports arbitrarily large parameter sets.

This work uses a 2× smaller RNS moduli compared to the original RPU paper [15]. Here, we assume 128 HPLEs and have modified the RPU's associated instruction set architecture (ISA), `B512`, to support a vector length of $1K$, referred to as `B1K`, to maintain high throughput and keep compute units occupied. Longer vectors make hardware efficient, e.g., taking pressure off the frontend and improving compute utilization. `B1K` consists of 28 instructions ranging from general purpose point-wise arithmetic operations to HE-specific *shuffle* instructions for (i)NTT kernels.

Since FHE is data-oblivious, all memory addresses are known at compile time, and the behavior is independent of input values. This property enables data prefetching through decoupling, where the compiler can order memory requests to overlap data movement with computation. This optimization leverages the deeply decoupled microarchitecture of the RPU, which employs three distinct queues to fetch independent compute, shuffle, and memory instructions in parallel. This overlap helps mask latency and improve overall performance.

B. Benchmark Selection

We use the RNS variant of HKS, as presented in Han [18], which optimizes the key-switching implementation by leveraging RNS decomposition for enhanced efficiency. The security

Table. III: Parameters satisfying 128-bit security.

Benchmark	N	k_l	k_p	dnum	α	evk Size	Temp data
BTS1	2^{17}	28	28	1	28	112MB	196MB
BTS2	2^{17}	40	20	2	20	240MB	400MB
BTS3	2^{17}	45	15	3	15	360MB	585MB
ARK	2^{16}	24	6	4	6	120MB	192MB
DPRIVE	2^{16}	26	7	3	9	99MB	163MB

of HE is defined by λ, which is a function of $\frac{N}{logQP}$. As mentioned in Cheon [27], 128-bit security is required to defend against dual attacks and to be able to extract the real data from the encrypted output. BTS [16] simulates the multiplication time per slot by sweeping N, L, and λ to identify the optimal configurations for 128-bit security, and refers to these points as BTS1, BTS2, and BTS3. ARK [17] uses a smaller polynomial degree, 2^{16}, while increasing the number of digits to 4 to satisfy the 128-bit security. As in recent studies [16], [17], [21], [28] we provide 128-bit security using proper FHE parameterization from BTS [16], ARK [17], and DPRIVE [29], to evaluate the performance of our dataflows. Table III summarizes the parameters of the benchmarks used in our research.

C. Software Framework

To generate HKS code for different benchmarks and dataflows, we break down the HKS algorithm into steps highlighted in Figure 1 and generate instructions for each step independently, based on the `B1K` ISA. Our simulation framework includes two distinct tasks: memory and compute. Memory tasks handle data transfer between off-chip and on-chip, while compute tasks correspond to steps in the HKS algorithm, executed on the RPU [15]. The challenge of our approach lies in effectively managing task dependencies and the sequence of instructions. Each task may rely on one or more compute and memory tasks, which must complete their execution before the dependent task can proceed. These dependencies stem from a variety of reasons, such as the need to fetch data from off-chip memory or to create space for subsequent operations using a memory task. Moreover, some tasks depend on data generated by a compute task as part of their input. These dependencies may vary based on the on-chip memory, the specific dataflow, and the benchmarks used. Using the software framework, we generate instructions for each configuration and dataflow and define their dependencies. The framework has two distinct queues, one for memory tasks and one for compute tasks. The tasks at the front of each queue are fetched and executed in parallel once all the task's dependencies are resolved. If there are no dependencies between a memory task and a compute task, the off-chip data movement can be masked by the computation on the RPU.

D. Dataflow Simulation

We have implemented and analyzed all given benchmarks in Table III for each dataflow described in Section IV. In Section IV, we noted that with unlimited on-chip memory, the performance of the dataflows tends to be nearly identical. However, with limited on-chip memory, this would no longer be true because the inputs and intermediate data cannot all fit on the chip at once. This makes the sequence of operations and the data movement between on and off-chip memory crucial. When dealing with small on-chip memory and increased intermediate data, the performance of DC may become close to MP due to more interactions with off-chip memory.

Our goal is to reduce interactions with off-chip memory by increasing on-chip data reuse. In general, when sufficient on-chip memory is available, INTT outputs can be stored on-chip and reused. However, with limited on-chip storage, the INTT output must be stored off-chip and reloaded for subsequent computations, increasing the off-chip data movement. For example, in BTS3, using the MP implementation, INTT is applied to all 45 input towers, leaving no space for storing all INTT outputs on-chip. In contrast, using OC, INTT is applied to 30 towers, allowing the INTT outputs to be stored on-chip and used for later computation.

In *ModUp_P5*, if on-chip space is sufficient, we prioritize storing towers related to $[P_0]_{\mathcal{B}}$ and $[P_1]_{\mathcal{B}}$, for optimized on-chip memory use and reduced off-chip memory access during subsequent computations. Additionally, *ModDown_P2* and *ModDown_P3* will begin computations from towers that are already on-chip.

VI. Evaluation

In this section, we evaluate HKS performance following the three dataflows described above, using five benchmarks listed in Table III from recent work (BTS [16], ARK [17], and DPRIVE [29]). The evaluation is done using the RPU [15] as the target hardware platform of our implementations, including the modifications mentioned in Section V-A. While the performance results are specific to the RPU, the general insights and takeaways from the optimizations are broadly applicable. As mentioned earlier, one of the challenges in HKS is properly handling large amounts of input, intermediate data, and $\mathbf{evks}$. In fact, prior work [16], [17], [30] has found that HE is memory bound. We re-examine this performance limitation considering the different proposed dataflows. First, we assume a large 392MB on-chip SRAM, with 32MB allocated for data and the rest for $\mathbf{evks}$, pre-loaded on-chip. We sweep bandwidth across all three dataflows to demonstrate the bandwidth saving achievable with OC while maintaining performance equal to MP and DC at higher bandwidths. Next, we eliminate the on-chip SRAM dedicated to storing $\mathbf{evks}$, instead streaming them on-chip, as they are used only once per HKS, and saving substantial on-chip SRAM (up to $12.25\times$). We find that a small on-chip memory significantly degrades the performance of a naive implementation (MP), but that our optimizations (OC) result in minimal slowdown. Finally, we increase the computational throughput of the RPU to understand how our proposed dataflow balances the communication-to-compute ratio.

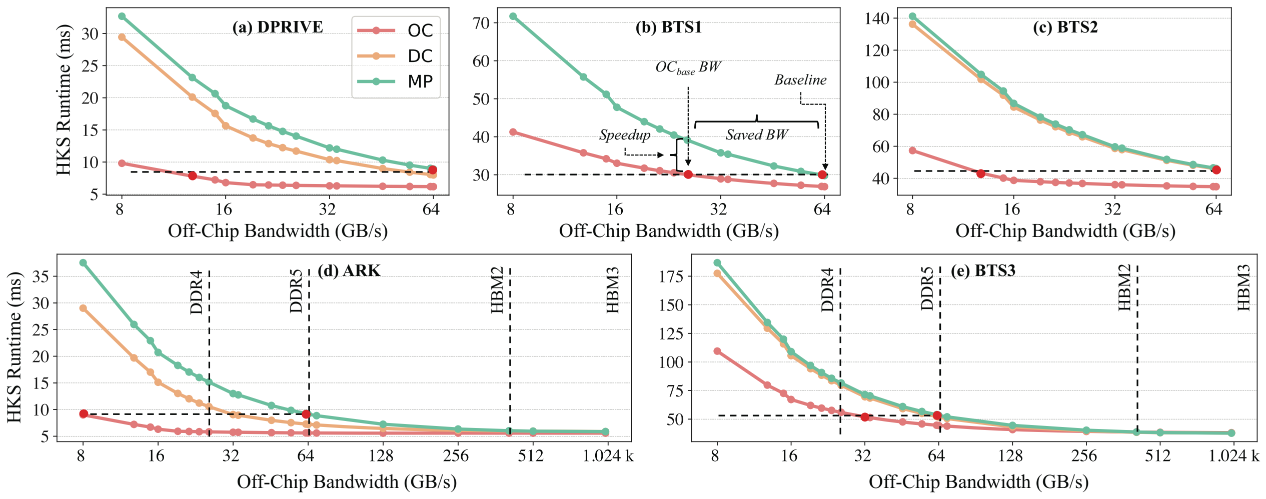

Figure. 4: Quantifying latency reduction for MP, DC, and OC by increasing DRAM bandwidth for the five given benchmarks.

A. *Dataflow Comparison: Saving Bandwidth*

Figure 4 compares different HKS implementation runtimes across different dataflows for all benchmarks. Benchmarks are categorized into three groups: ARK and DPRIVE (small, 2^{16} polynomial degree); BTS1 (single digit, lacks *ModUp_Reduce*), and BTS2 and BTS3 (large, 2^{17} with three and four digits). We sweep the off-chip bandwidth from 8GB/s to 64GB/s, extending to 1TB/s for ARK and BTS3, representing the smallest and largest benchmarks, respectively. This bandwidth range includes DDR4 (8GB/s to 25.6GB/s), DDR5 (32GB/s to 64GB/s), HBM2 (64GB/s to 410GB/s), and HBM3 (up to 1TB/s). In all benchmarks, the performance benefit of OC is large at low bandwidth and decreases as bandwidth increases. This is the result of RPU becoming compute bound, reducing the impact of off-chip communication. With OC, we show that by optimizing the dataflow we can achieve higher performance with less bandwidth.

Below we will delve into a detailed comparison between OC, MP, and DC runtimes, taking the MP implementation with 64GB/s off-chip bandwidth and on-chip pre-loaded **evks** as our *baseline*. We will explore the bandwidth at which OC matches *baseline* performance, denoting it as OC_{base}. 64GB/s was chosen as our reference point, given that it represents the peak DDR5 bandwidth in our evaluations, and highlighting cost-effective designs before moving to expensive memory technologies.

1) DPRIVE & ARK: Figure 4(a) and (d), demonstrate a significant improvement in HKS runtime with OC dataflow. The horizontal line in Figure 4(a) shows that OC operating at 12.8GB/s bandwidth matches the baseline, resulting in a 5× bandwidth saving. At this specific data point, OC is 2.57× and 2.96× faster than DC and MP, respectively. Before becoming compute bound, the RPU is idle for part of its execution time, waiting for dependent memory tasks to be completed. With 12.8GB/s off-chip bandwidth, OC causes the RPU to be idle for 20.87% of its execution time, outperforming DC and MP with 68.62% and 72.76% idle times, respectively. This highlights OC's advantages when off-chip bandwidth is a concern and its efficiency in minimizing idle time.

In the case of ARK, the OC implementation with 8GB/s off-chip bandwidth achieves the same performance as MP and DC with 64GB/s bandwidth, resulting in 8× and 5× bandwidth savings, respectively. With 8GB/s off-chip bandwidth, OC outperforms MP and DC runtime by notable factors of 4.16× and 3.22× and being 2.25× less idle cycle than MP.

2) BTS1: As mentioned above, for BTS1 with one digit, MP and DC have the same implementation. In Figure 4(b), OC matches the baseline performance by 25.6GB/s off-chip bandwidth, saving 2.5× bandwidth. At this point, OC is 1.3× faster than MP and 2.1× less idle.

3) BTS2 & BTS3: By analyzing BTS2 and BTS3, we observe that as the benchmark size increases, DC and MP converge due to the large data size during *BConv* expansion and more polynomial additions for reduction.

According to Figure 4(c), BTS2's runtime with OC dataflow and less than 12.8GB/s off-chip bandwidth matches the baseline, saving 5× bandwidth. With 12.8GB/s bandwidth, OC is 2.35× faster than DC and 2.42× faster than MP.

For BTS3, the performance gap between OC and the other two dataflows decreases as the RPU becomes more compute bound. Despite the decreasing gap between OC and MP/DC, due to the increased input and intermediate data in BTS3, OC still outperforms MP/DC. With 32GB/s bandwidth, OC is 1.3× faster and 1.7× less idle than MP and DC, while matching the baseline's performance, resulting in a 2× bandwidth saving.

4) Comparison Across Benchmarks: Table IV summarizes the previous discussion, with "OC_{base}" indicating the bandwidth where OC matches the *baseline* performance (MP with 64MB/s bandwidth), and "*Saved BW*" showing the bandwidth savings achieved by OC. Runtimes are reported based on the OC_{base} bandwidth. Notably, BTS1 has less speedup due to its larger size compared to ARK and DPRIVE and more data movement for the reduction than BTS2 and BTS3. This can also be seen in Table II, where the AI (Arithmetic Intensity)

Table. IV: OC bandwidth (OC_{base}; in GB/s) required to achieve the same performance as MP using 64GB/s BW. Speedup reports the performance improvement OC achieves over MP at the listed bandwidth (OC_{base}).

Benchmark	$\mathbf{OC_{base}}$ (GB/s)	Saved BW	OC (ms)	MP (ms)	OC Speedup
BTS1	25.60	2.50x	30.08	39.13	1.30x
BTS2	12.80	5.00x	43.24	104.85	2.42x
BTS3	32.00	2.00x	51.87	71.50	1.37x
ARK	8.00	8.00x	9.01	37.54	4.16x
DPRIVE	12.80	5.00x	7.81	23.15	2.96x

improvement for BTS1 in OC is less than other benchmarks.

Many recent HE algorithmic optimizations, e.g., RNS-based HKS [18], which is used in this paper, require a higher bandwidth. Therefore, reducing off-chip bandwidth is crucial. The OC dataflow results in 1.30× to 4.16× speedup over a simple MP implementation and saves 2× to 8× off-chip bandwidth.

B. *Streaming Evaluation Keys: Trading SRAM for Bandwidth*

Analyzing the computations of HKS, the input data and intermediate data (*BConv* output) are reused for calculating multiple output towers, making it advantageous to keep them on-chip to reduce off-chip communication. On the other hand, the **evks** are large, ranging from 99MB (DPRIVE) to 360MB (BTS3), and are only used once per HKS. Consequently, it can be more practical and area efficient to store these large **evks** off-chip and load them on-chip in a streaming fashion. To study the effects of streaming **evks**, we reserve a fraction of off-chip bandwidth and dedicate it to loading the **evks** on-chip. In this case, we only have 32MB of on-chip memory to store data and capture on-chip reuse. To determine the required bandwidth allocation for **evks**, we calculate the ratio of the number of **evks** to the number of loaded/stored data.

Figures 5 and 6 illustrate the effects of storing **evks** off-chip and HKS runtime as a function of bandwidth. The dotted lines represent the runtime with pre-loaded **evks** stored on-chip. As shown in the figures, storing **evks** off-chip maintains the same trend but with shifted absolute values due to the increased bandwidth pressure from streaming **evks**. According to Table IV, for BTS3, 32GB/s is the OC_{base} bandwidth (equal performance as baseline). However, Figure 5 demonstrates that by streaming **evks** on-chip, the OC dataflow can achieve baseline performance with a higher bandwidth of 45.62GB/s. Similarly, Figure 6 shows this concept for ARK, where a bandwidth of 23.4GB/s allows ARK to match baseline performance. While in both these cases more bandwidth is required to match performance, we argue the increase is minor compared to the 12.25× reduction in on-chip SRAM.

Figure 7 shows the slowdown for the OC dataflow of each benchmark when streaming **evks** from off-chip. The results are shown for two bandwidths per benchmark, as indicated by the clustered bars (e.g., 8 and 23.4 for ARK). The first bandwidth corresponds to the OC_{base} bandwidth specified in

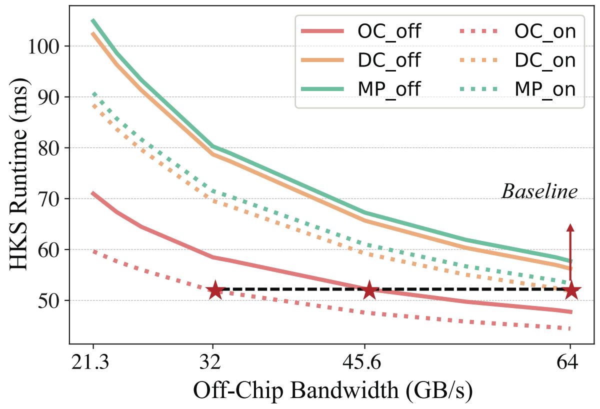

Figure. 5: HKS runtime for BTS3 with **evks** being off-chip.

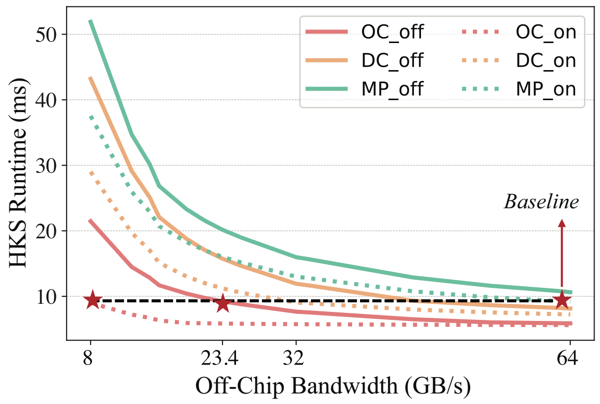

Figure. 6: HKS runtime for ARK with **evks** being off-chip.

Table IV, where the performance is equal to the baseline, assuming **evks** to be on-chip. The second bandwidth indicates the required bandwidth to attain equivalent performance when streaming the **evks** from off-chip. The slowdown for DPRIVE is less than ARK due to the smaller ratio of **evks** to the loaded/stored data (0.66 for ARK and 0.5 for DPRIVE). Among the large benchmarks (BTS1, BTS2, and BTS3), BTS2 has the most slowdown, 1.33×, since the mentioned ratio is more than other benchmarks. As shown in Figure 7, with the OC dataflow, and 1.3× (BTS1) to 2.9× (ARK) more bandwidth we can achieve the same performance as storing the **evks** on-chip while saving 12.25× on-chip memory. However, compared to the original 64GB/s MP implementation with **evks** on-chip, the OC dataflow with keys off-chip still saves 1.4× up to 3.3× bandwidth for BTS3 and BTS2, respectively, to achieve the same performance. In conclusion, storing **evks** off-chip saves 12.25× on-chip memory and by implementing OC dataflow, we can save up to 3.3× bandwidth to have the same performance as the baseline. Additionally, storing **evks** off-chip and keeping 32MB on-chip memory for data, decreases the RPU area from $401.85mm^2$ to $41.85mm^2$.

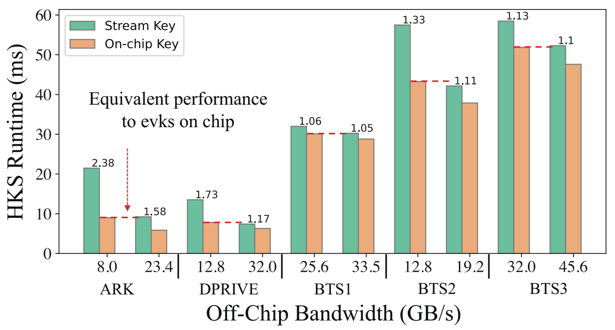

Figure. 7: HKS runtime for OC with **evks** streamed from off-chip vs. **evks** stored on-chip, and the equivalent bandwidth.

C. Sensitivity Analysis: Balancing Bandwidth and Computational Throughput

In the last section, we evaluated the three proposed dataflows and showed how data reuse enabled the OC optimization to significantly improve the runtime of HKS, saving considerable bandwidth even when streaming **evks**. In this section we conduct a sensitivity study to investigate how the performance of HKS with OC is affected as we increase both the bandwidth, ranging up to 1TB/s (HBM3), and the computational throughput by up to 16× more. We evaluate HKS performance for ARK and BTS3 benchmarks, the smallest and largest benchmarks, respectively. We refer to computational throughput as MODOPS (Modular Operations per Second).

1) Bandwidth Analysis: In Figure 4(d) and (e) we increase the off-chip bandwidth beyond 64GB/s (DDR5) up to 1TB/s (HBM3), with HPLEs fixed at 128, to study how scaling bandwidth affects HKS runtime for ARK and BTS3. For both ARK and BTS3, the benefit from OC compared to the other two dataflows diminishes at bandwidths larger than 256GB/s. At this point, the off-chip data movement is mostly masked by computation on the RPU, limiting further performance gain from increasing bandwidth.

According to Figure 4(e), moving from 1TB/s to the OC_{base} BW for BTS3, results in a 31.25× bandwidth saving, with only a 1.35× increase in runtime, while transitioning to the same bandwidth (32GB/s) with the MP dataflow leads to a 13.98× slower runtime than the 1TB/s implementation.

In Figure 4(d), we see that by employing OC dataflow for ARK, data movement becomes fully masked by computation after reaching a bandwidth of 128GB/s. With 8GB/s of off-chip bandwidth (OC_{base}) and the OC implementation, we can save 16× bandwidth by being 1.6× slower compared to OC with 128GB/s. That is while for MP implementation, moving to 8GB/s causes a 5.17× increment in the runtime.

2) Computational Throughput Analysis: The RPU design includes 128 lanes, meaning 128 modular multipliers, which is 128× less functional units compared to BTS [16]. To enhance the HKS runtime, which benefits from parallel computation where the calculation of each output tower is entirely independent of others, we simulate an accelerator with greater computational throughput by increasing the RPU's MODOPS

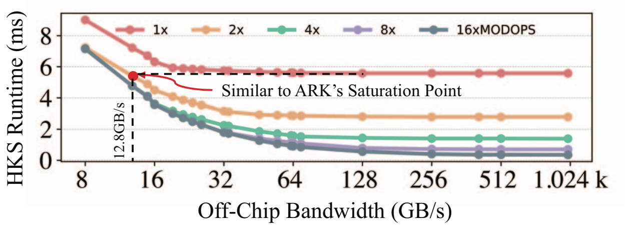

Figure. 8: Evaluating HKS runtimes for ARK using OC at different MODOPS, with evks on-chip.

Table. V: OC, DC, and MP configuration for equivalent performance to ARK's saturation point.

Dataflow	BW GB/s	RPU MODOPS	Rel. BW	Rel. MODOPS
Sat. Point	128	1.00x	1.00x	1.00x
OC	12.80	2.00x	0.10x	2.00x
DC	54.64	2.00x	0.42x	2.00x
MP	128.00	2.00x	1.00x	2.00x

(2×, 4×, 8×, and 16×). We will see how increasing the MODOPS will impact ARK's OC performance. First, we will consider **evks** to be pre-loaded on-chip.

Based on Figure 8, at low bandwidths, the HKS runtime for different MODOPS is nearly identical, as they are bandwidth limited and do not benefit from increased computational throughput. However, at higher bandwidths, HKS becomes compute bound, resulting in an increasing gap between the HKS runtime at different MODOPS. As mentioned earlier, in ARK's OC implementation, off-chip data movement is entirely masked by computation at 128GB/s. We will call this point "*ARK's saturation point*". At 128GB/s, the design is no longer limited by bandwidth. Doubling the MODOPS reduces the HKS runtime. Therefore, as shown in Figure 8, saturation performance can be achieved with less bandwidth (12.8GB/s) with 2× MODOPS, saving 10× on-chip bandwidth.

Table V outlines the required bandwidth and computational throughput for DC and MP to match ARK's saturation point. Maintaining the same MODOPS as OC necessitates at least 4.26× and 10× higher bandwidth for DC and MP, respectively.

Figure 9 shows the required MODOPS and bandwidth to get the same performance as "*ARK's saturation point*" and the "*baseline*" with OC while streaming the **evks** on-chip and having a 32MB on-chip memory. We previously showed that 12.8GB/s bandwidth with 2×MODOPS has the same performance as ARK's saturation point, with **evks** being on-chip. To get the same performance while streaming the **evks**, 2.6× more bandwidth with the same MODOPS (2×) is required, saving 12.25× on-chip memory. However, with 1×MODOPS, 20× more bandwidth is required to get the saturation performance. At 256GB/s the design is completely compute bound; therefore, we can find a better balance between bandwidth and MODOPS by increasing the MODOPS

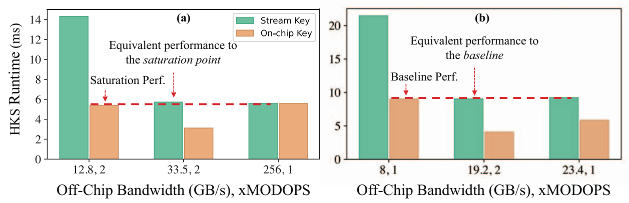

Figure. 9: ARK's configuration for equivalent performance to the baseline and saturation point with OC and streaming evks.

and decreasing the bandwidth. This intuition also applies to the *baseline*. As Figure 9(b) shows, by doubling the MODOPS we can achieve the same performance as 1×MODOPS and save 1.2× bandwidth.

VII. Related Work

High-Performance CPU Implementations: Many software libraries exist that support HE [5]–[8]. Notably, Lattigo [7], OpenFHE [6], and HEAAN [5] support CKKS bootstrapping and FHE. However, due to limited computational power, some applications like deep neural network inference, remain impractical on pure CPU implementations. For instance, state-of-the-art ResNet models [19], [31], [32] can take upwards of one hour *per* inference, limiting their applicability to infrequent, low arrival rate applications [33], [34].

Compilers and Dataflow: Recently, many works have focused on FHE compiler development [26], [35]–[42], particularly for private neural inference. Tools such as CHET [35] and EVA [36] automatically map and optimize common neural network layers for FHE, managing parameter selection and ciphertexts implicitly. HECO [37] is an end-to-end compiler that converts high-level programs into secure FHE circuits, allowing non-experts to develop secure and efficient FHE applications. Porcupine [39] and Coyote [42] generate vectorized HE code for small HE kernels, with Coyote [42] optimizing data layout to minimize the number of rotations. Orion [38] utilizes double-hoisting [43] to enhance CPU latencies. MAD [26] highlights how caching techniques and dataflow optimizations can yield high performance bootstrapping implementations despite small on-chip memories (1 to 32 MB). While MAD [26] presents a dataflow similar to our DC approach, our OC strategy, using 32 and 392MB of on-chip memory, further increases the arithmetic intensity of key-switching by leveraging on-chip data reuse.

GPU/FPGA Acceleration: GPUs offer performance gains with their numerous parallel compute units and high off-chip bandwidth. Previous studies explore GPU acceleration of HE operations [12], [44]–[48]. Jung [46] was the first the support CKKS on GPU, suggesting optimizations like *kernel fusing* to reduce on-chip memory requirements. Still, the lack of native modular arithmetic support and limited on-chip memory yields subpar performance compared to modern ASIC solutions [14], [16], [17], [23].

Another approach is to use FPGAs [13], [21], [49]–[51]. While FPGAs may not provide the same level of performance as ASIC solutions, their re-programmability is advantageous as FHE algorithms evolve. HEAX [13] is an FPGA-based accelerator for FHE CKKS. Recently, FAB [21] and Poseidon [49] proposed FPGA-based accelerators to support bootstrapping. FAB [21] proposes several dataflow optimizations, alongside an FPGA implementation, geared towards increasing on-chip data reuse and minimizing unnecessary DRAM accesses.

ASIC Acceleration: Cheetah [9] was the first to present a large-scale ASIC with custom logic and show that the overheads of HE could be overcome with hardware acceleration. `F1` [10] presented a programmable ASIC accelerator for FHE but was tailored to small parameter sets with limited bootstrapping support, making it inefficient for *deep* neural network computations. Since then, many works [14], [16], [17], [23] proposed ASIC solutions targeting bootstrapping and FHE. CraterLake [14] and BTS [16] were the first ASIC accelerators supporting bootstrapping, but featured underutilized compute units despite allocating 1TB of off-chip bandwidth. Later, ARK [17] proposed a minimal key-switching strategy to address memory bottlenecks, yet still required a large 512MB on-chip scratchpad. Recently, SHARP [23] characterized the effect of smaller moduli size on neural network accuracy, demonstrating the feasibility of using smaller 36-bit RNS moduli to reduce compute unit size, on-chip memory requirements, and off-chip bandwidth pressure.

VIII. Conclusion

In this paper, we propose three distinct approaches for implementing the hybrid key-switching (HKS) algorithm that differ in the sequence of instructions and data reuse strategies. The large amount of **evks** and generated intermediate data (up to 1.5GB) puts pressure on implementing FHE applications. We introduce a novel dataflow, named Output-Centric (OC), that reduces the off-chip data movement and increases data reuse. Through our evaluations using the RPU accelerator with various benchmarks, our OC dataflow achieves up to 4.16× speedup over a naive Max-Parallel (MP) implementation of HKS. Additionally, we can save 12.25× of on-chip SRAM by storing the **evks** off-chip and save 3.3× bandwidth with up to 2.4× more arithmetic intensity compared to a MP on-chip implementation.

Acknowledgment

Support for this work was provided in part by NSF CAREER award #2340137. This work was supported in part by Graduate Assistance in Areas of National Need (GAANN) and funding from DARPA, under the Data Protection in Virtual Environments (DPRIVE) program, contract HR0011-21-9-0003. Reagen and Ebel received generous support from the NY State Center for Advanced Technology in Telecommunications (CATT) and a gift award from Google. We also express our gratitude to Duality, especially David Bruce Cousins, Ahmad Al Badawi, and Yuriy Polyakov as well as Andrew G. Schmidt from AMD for their valuable insights and support. The views, opinions, and/or findings expressed are those of the authors and do not necessarily reflect the views of sponsors.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

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

[3] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual Cryptology Conference*, pp. 868–886, Springer, 2012.

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

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pp. 409–437, Springer, 2017.

[6] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library." Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[7] "Lattigo 1.3.0." Online: http://github.com/ldsec/lattigo, Dec. 2019. EPFL-LDS.

[8] "Microsoft SEAL (release 4.1)." https://github.com/Microsoft/SEAL, Jan. 2023. Microsoft Research, Redmond, WA.

[9] 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 26–39, IEEE, 2021.

[10] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 238–252, 2021.

[11] A. Aikata, A. C. Mert, S. Kwon, M. Deryabin, and S. S. Roy, "Reed: Chiplet-based scalable hardware accelerator for fully homomorphic encryption," *arXiv preprint arXiv:2308.02885*, 2023.

[12] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," 2022.

[13] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*, pp. 1295–1309, 2020.

[14] 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*, pp. 173–187, 2022.

[15] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. Al Badawi, Y. Polyakov, K. Canida, *et al.*, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 272–282, IEEE, 2023.

[16] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 711–725, 2022.

[17] 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)*, pp. 1237–1254, IEEE, 2022.

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

[19] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *Proceedings of the 39th International Conference on Machine Learning* (K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, eds.), vol. 162 of *Proceedings of Machine Learning Research*, pp. 12403–12422, PMLR, 17–23 Jul 2022.

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

[21] 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)*, pp. 882–895, 2023.

[22] 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: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*, pp. 347–368, Springer, 2019.

[23] 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 *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[24] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," pp. 6. 64–70, 2020.

[25] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library." Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[26] 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 *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, (New York, NY, USA), p. 685–697, Association for Computing Machinery, 2023.

[27] J. H. Cheon, M. Hhan, S. Hong, and Y. Son, "A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret lwe," *IEEE Access*, vol. 7, pp. 89497–89506, 2019.

[28] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 587–617, Springer, 2021.

[29] "DARPA DPRIVE Broad Agency Announcement (BAA), HR001120S0032." https://www.dla.mil/Strategic-Materials/Business/Broad-Agency-Announcement/ Accessed on Nov-01-2023.

[30] 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.

[31] D. Kim, J. Park, J. Kim, S. Kim, and J. H. Ahn, "Hyphen: A hybrid packing method and optimizations for homomorphic encryption-based neural networks," 2023.

[32] D. Kim and C. Guyot, "Optimized privacy-preserving cnn inference with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2175–2187, 2023.

[33] K. Garimella, Z. Ghodsi, N. K. Jha, S. Garg, and B. Reagen, "Characterizing and optimizing end-to-end systems for private inference," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ACM, mar 2023.

[34] J. Mo, K. Garimella, N. Neda, A. Ebel, and B. Reagen, "Towards Fast and Scalable Private Inference," *arXiv e-prints*, p. arXiv:2307.04077, July 2023.

[35] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: compiler and runtime for homomorphic evaluation of tensor programs," *arXiv preprint arXiv:1810.00845*, 2018.

[36] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, pp. 546–561, 2020.

[37] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, “{HECO}: Fully homomorphic encryption compiler,” in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 4715–4732, 2023.

[38] A. Ebel, K. Garimella, and B. Reagen, “Orion: A fully homomorphic encryption compiler for private deep neural network inference,” 2023.

[39] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, “Porcupine: A synthesizing compiler for vectorized homomorphic encryption,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 375–389, 2021.

[40] Zama, “Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists,” 2022. https://github.com/zama-ai/concrete-ml.

[41] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, “Hecate: Performance-aware scale optimization for homomorphic encryption compiler,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–204, IEEE, 2022.

[42] R. Malik, K. Sheth, and M. Kulkarni, “Coyote: A compiler for vectorizing encrypted arithmetic circuits,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 118–133, 2023.

[43] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, “Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys.” Cryptology ePrint Archive, Paper 2020/1203, 2020. https://eprint.iacr.org/2020/1203.

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

[45] S. Kim, W. Jung, J. Park, and J. H. Ahn, “Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, oct 2020.

[46] 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*, vol. 2021, p. 114–148, Aug. 2021.

[47] Y. B. Kaustubh Shivdikar, M. S. Rashmi Agrawal, E. M. Gilbert Jonatan, J. L. Abellán, A. Ingare, J. K. Neal Livesay, and D. K. Ajay Joshi, “Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption,” *MICRO’23, October 28-November 1, 2023, Toronto, ON, Canada*, 2023.

[48] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellán, J. Kim, and D. Kaeli, “Accelerating polynomial multiplication for homomorphic encryption on gpus,” in *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pp. 61–72, IEEE, 2022.

[49] 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)*, pp. 870–881, 2023.

[50] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, “An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus,” in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021.

[51] 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)*, pp. 896–907, IEEE, 2023.