HMC-FHE: A Heterogeneous Near Data Processing Framework for Homomorphic Encryption

Zehao Chen, Zhining Cao, Zhaoyan Shen, and Lei Ju

***Abstract*—Fully homomorphic encryption (FHE) offers a promising solution to ensure data privacy by enabling computations directly on encrypted data. However, its notorious performance degradation severely limits the practical application, due to the explosion of both the ciphertext volume and computation. In this article, leveraging the diversity of computing power and memory bandwidth requirements of FHE operations, we present HMC-FHE, a robust acceleration framework that combines both GPU and hybrid memory cube (HMC) processing engines to accelerate FHE applications cooperatively. HMC-FHE incorporates four key hardware/software co-design techniques: 1) a fine-grained kernel offloading mechanism to efficiently offload FHE operations to relevant processing engines; 2) a ciphertext partitioning scheme to minimize data transfer across decentralized HMC processing engines; 3) an FHE operation pipeline scheme to facilitate pipelined execution between GPU and HMC engines; and 4) a kernel tuning scheme to guarantee the parallelism of GPU and HMC engines. We demonstrate that the GPU-HMC architecture with proper resource management serves as a promising acceleration scheme for memory-intensive FHE operations. Compared with the state-of-the-art GPU-based acceleration scheme, the proposed framework achieves up to 2.65× performance gains and reduces 1.81× energy consumption with the same peak computation capacity.**

***Index Terms*—Accelerator, homomorphic encryption, near memory processing.**

I. Introduction

FULLY homomorphic encryption (FHE) is an encryption scheme that enables computations to be performed directly on encrypted data, where no decryption for intermediate steps during the computation is required. With the ability to perform arithmetic operations without revealing user data privacy, FHE becomes one of the most promising privacy protection techniques and has been gradually deployed in application scenarios, including encrypted databases [1] and machine learning [2].

Nonetheless, the concerning computational speed of FHE remains a major obstacle hindering its rapid advancement. Since the method of constructing the ciphertext space results in a substantial expansion of computation and data scale, the computing speed of ciphertext experiences a drastic degradation of several orders of magnitudes ($10^3 \sim 10^6$) compared to plaintext [3]. To solve this issue, numerous studies [4], [5], [6], [7], [8], [9], [10], [11] have focused on building domain-specific hardware accelerators to enhance the performance of FHE operations through resource reuse and increased parallelism. Meanwhile, researchers have also explored deploying FHE operations on GPU platforms to support privacy protection applications in various scenarios comprehensively [12], [13], [14], [15]. These efforts have yielded a substantial enhancement in the speed of ciphertext computation, reducing the performance gap by 2∼3 orders of magnitude.

In the design of FHE accelerators, due to the memory-intensive nature, the demand for memory capacity and bandwidth far surpasses the computational requisites. For instance, ASIC schemes typically use large on-chip storage [7], [16]. Meanwhile, FPGA (or GPU)-based accelerator designs identify on-chip BRAM (or Shared Memory) capacity as the performance bottleneck [4], [12]. In this work, we further conduct extensive quantitative analysis on the arithmetic intensity (AI) of each FHE operation on GPU devices. Our analysis results further reinforce the significance of prioritizing the alleviation of bandwidth constraints as the foremost endeavor in accelerating FHE operations.

The memory-intensive characteristics of FHE naturally make hybrid memory cube (HMC) a natural choice for FHE acceleration. In particular, the GPU-HMC architecture (as demonstrated in previous work [17], [18], [19], [20], [21]) as shown in Fig. 1 is a promising candidate which is composed of a centralized GPU device and multiple auxiliary HMC processing engines. By encapsulating various computing logic for computing while having the characteristics of high-internal bandwidth, low latency, and low-power consumption, the GPU-HMC architecture offers a balanced tradeoff between computing power and bandwidth resources for individual FHE operations. Nonetheless, relying solely on this architectural

Manuscript received 10 August 2024; accepted 10 August 2024. Date of current version 6 November 2024. This work was supported in part by the Natural Science Foundation of China under Grant 62372272; in part by the Department of Science and Technology of Shandong Province under Grant SYS202201; in part by the Quan Cheng Laboratory under Grant QCLZD202302 and Grant CCF-AFSG RF20220011; in part by the Taishan Scholars Program under Grant tsqn202211281; in part by the Key Research and Development Program of Shandong Province under Grant 2023CXPT002; and in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Grant 2022WNLOKF018. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Lei Ju.)*

Zehao Chen is with the School of Cyber Science and Technology, Shandong University, Qingdao 266237, China, and also with Quan Cheng Laboratory, Jinan 250012, China.

Zhining Cao is with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China.

Zhaoyan Shen is with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China, and also with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China.

Lei Ju is with Quan Cheng Laboratory, Jinan 250012, China (e-mail: julei@sdu.edu.cn).

Digital Object Identifier 10.1109/TCAD.2024.3447212

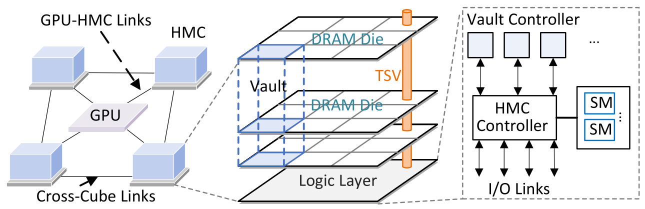


Fig. 1. Typical GPU-HMC architecture [17], [18].

setup leads to suboptimal overall performance enhancement, and several challenges must be effectively addressed with an automatic design flow.

Challenge 1: Different FHE operations exhibit varying computation and memory intensity. Thus, exquisitely offloading FHE operations to their preferred processing engines (GPU or HMCs) becomes a primary problem to address.

Challenge 2: To execute an FHE operation on distributed HMC processing engines in a parallel fashion, it is essential to reduce the inter-HMC data transfer between parallel executed subtasks.

Challenge 3: It is critical to utilize the parallel computation capacity between GPU and HMCs for overall system performance, which necessitates sophisticated scheduling to achieve inter- and intra-operation parallelism.

In this article, to overcome the above challenges, we propose a design flow framework for GPU-HMC-based FHE acceleration, which consists of a series of hardware/software co-designs. First, we introduce an offloading mechanism for FHE operations, which categorizes the operations into reusable basic kernels and determines the affinity for the GPU or HMC engines based on the AI. Moreover, we propose a ciphertext partitioning scheme by decoupling the structure of ciphertext polynomials, which ensures efficient global memory bandwidth utilization while achieving parallelism and load balancing. Then, we undertake a thorough analysis of data dependency and propose an interoperation pipeline scheme to facilitate parallel execution between GPU and HMC engines. Finally, for HE operations outside the above-mentioned pipeline stages, we propose a fine-grained intraoperation tuning scheme to further balance the workload between GPU and HMC engines.

The contributions of this work are summarized as follows.

1) To the best of the authors knowledge, this is a pioneer work that introduces FHE operation acceleration with heterogeneous GPU-HMC architecture. It unveals that GPU-HMC architecture is a promising design choice for FHE acceleration given the distinct computation and memory intensity between FHE operations.
2) This article proposes a design flow framework which automatically performs FHE operation scheduling and data allocation on the GPU-HMC architecture. The framework synergistically integrates a set of hardware/software co-design techniques to achieve working balancing and high-memory bandwidth utilization.
3) The experimental results on practical FHE workloads show up-to 2.65× speedup and up-to 1.81× energy efficiency (EE), compared to the state-of-the-art GPU-based accelerator design. Meanwhile, the scalability of the proposed GPU-HMC architecture has also been evaluated.

The remainder of this article is organized as follows. Section II introduces the background and motivation. Section III gives an overview of HMC-FHE framework. Section IV describe the detail techniques of HMC-FHE. Section V evaluates HMC-FHE. Sections VI and VII discuss the related work and conclusion.

II. Background and Motivation

In this section, we first introduce the basic of FHE with a typical algorithm CKKS [22]. Next, we briefly introduce the structure of hybrid memory cube (HMC) processing engines. Lastly, we provide the motivation of this work.

A. FHE Schemes

FHE, exampled by schemes, such as CKKS [22], BFV [23], and BGV [24], enables arbitrary computations on ciphertexts without the need for decryption. This article primarily concentrates on the CKKS scheme due to its unique capability to handle plaintext inputs comprising arbitrary fixed-point real numbers, which has facilitated its application across a broader range of fields. Note that the proposed techniques and schemes are equally applicable to other FHE schemes, e.g., BFV, and BGV.

Ciphertext Structure in CKKS: In the CKKS scheme, each batch of $N/2$ real numbers is encoded and encrypted as a pair of polynomials with degree N in the ring $R_Q = Z_Q[X]/(X^N+1)$, where Q is a prime number with hundreds or thousands of bits that relate directly to the ciphertext spaces. Each polynomial consists of N coefficients, which are integers modulo by Q. To enable efficient modulo computations of the wide Q and coefficients, the residue number system (RNS) is presented to convert these wide coefficients into L residue polynomials with machine-word-length coefficients. Consequently, the ciphertext can be initially represented as a pair of 2-D matrices with width L and depth N.

CKKS Basic Kernel: The CKKS algorithm is built upon a set of basic kernels. All basic kernels are summarized as follows.

1) *Addition/Subtraction (Add/Sub)*, which execute element-wise operations to add or subtract two polynomials.
2) *Tensor Product (TensorP)*, which conducts element-wise product of two polynomials.
3) *Number Theory Transformation (NTT and iNTT)*, which enables the conversion between polynomials represented by coefficients and those represented by point-values, significantly speeding up polynomial multiplication.
4) *Fast Basic Conversion (Conv)*, which converts the RNS basis based on B $= \{p_0, p_1, \ldots, p_{K-1}\}$ and C $= \{q_0, q_1, \ldots, q_l\}, 0 \leq l \leq L$. We refer to the operations occurring during modulus increase or modulus decrease as *ConvUp (ConvU)* and *ConvDown (ConvD)*, respectively.
5) *Inner Product (InnerP)*, which integrates several *TensorP* kernels along with a single *Add* kernel to accumulate the results of *TensorP*.

TABLE I
DIFFERENT CKKS OPERATIONS

APIs	Description	Basic Kernels
CCAdd	Add two ciphertexts.	Add
PCMult	Multiply a plaintext with a ciphertext.	TensorP
CCMult	Multiply a ciphertext with a ciphertext.	TensorP, NTT, iNTT, Conv, InnerP
Rotate	Rotate the ciphertexts according to an index.	NTT, iNTT, Conv, Sub, InnerP, AMorph
HRotate	Batch processing of Rotate operation.	NTT, iNTT, Conv, Sub, InnerP, AMorph

6) *Automorph (AMorph)*, which performs the permutation operation on the polynomial using an index *r*.

CKKS Compound Kernel: The aforementioned basic kernels are typically amalgamated into more complex compound kernels, as outlined below.

1) *ModUp and ModDown* adjust the precision of the ciphertext modulus. *ModUp* enhances precision to accommodate more sophisticated CKKS operations, whereas *ModDown* lowers precision to control noise expansion. Both *ModUp* and *ModDown* encompass *iNTT*, *ConvU* (for *ModUp*)/*ConvD* (for *ModDown*), and *NTT* kernels.
2) *KeySwitch* facilitates the efficient and secure transition of ciphertexts encrypted at varying levels. This mechanism is crucial in CKKS for operations like *CCMult, Rotate*. Specifically, it consists of *ModUp*, *InnerP*, and *ModDown* kernels.
3) *Rescale* is designed to constrain noise within a specified range. It is composed of *iNTT*, *NTT*, and *Sub* kernels.
4) *Bootstrapping* mitigates the accumulated error throughout the computational process and resets the ciphertext's noise level, thereby enabling continued operations on the ciphertext. This operation encompasses all basic kernels.

CKKS APIs: By reorganizing these kernels based on specific rules, CKKS provides a series of APIs for external applications to implement homomorphic computation. Table I details the functional description and composition of kernels for these APIs.

B. Hybrid Memory Cube

Fig. 1 illustrates a heterogeneous GPU-HMC architecture with four HMCs, and each HMC adopts a 3-D-stacking architecture within a single package, layering multiple memory dies atop a logic die through the utilization of through-silicon via (TSV) technology. Each memory die is subdivided into several partitions, which are vertically aligned to form vaults, analogous to conventional memory channels. To enhance parallel processing across the vaults in the HMC, a dedicated vault controller is assigned to each vault. The logic inside an HMC typically incorporates one or more streaming multiprocessors (SMs) [25] or bespoke processing units [26], tailored to boost performance across a variety of applications. Data inside an HMC can be directly transferred to its logic layer via the TSV technology, allowing for short-path data transfer and, in combination with the parallel structure of vaults, achieving high-internal bandwidth and low-access latency, whereas since the number of processing units in the HMC logic layer is relatively limited, it offers reduced computational capacity compared to GPUs, making it generally suitable for memory-intensive workloads.

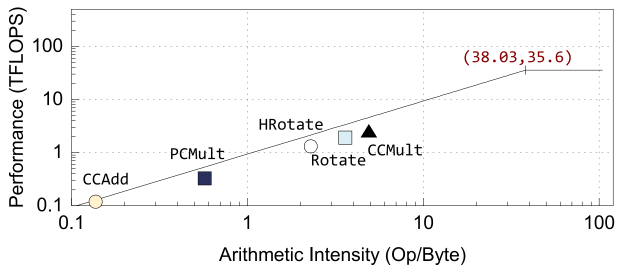


Fig. 2. Roofline model of different CKKS APIs on NVIDIA RTX Geforce 3090.

Moreover, HMC exhibits impressive scalability. Multiple HMC devices can be interconnected using I/O links on a silicon interposer, enabling parallel operations, or they can be connected to GPUs to facilitate joint processing efforts. As illustrated in Fig. 1, packet-based protocols are utilized for communications between HMCs or between HMCs and GPUs, supporting the bidirectional flow of commands and data. It should be noted that the latency for data transfer between HMC devices via I/O links is over three times higher than the latency for local access by the processing units within the logic layer.

C. Motivation

In this section, we first provide two observations of CKKS kernels with some preliminary experiments. Then, we discuss the advantages of a GPU-HMC heterogeneous architecture in enhancing the performance of FHE operations.

1) *Observation 1—CKKS APIs Exhibit a Greater Demand for Memory Bandwidth Compared to Computational Requirements:* The arithmetic logic of the CKKS scheme is based on the polynomial field of ciphertext, so it inherits the memory-intensive characteristic of polynomial computations. This highlights the reality that greater computational power is not the primary impetus for accelerating FHE schemes. Instead, the predominant constraint that curtails the performance is the efficiency of data movement. To further quantify the impact of bandwidth on performance and to gain insight into the memory/compute characteristics of FHE operations, we build a roofline model [27] using the NVIDIA GeForce RTX 3090 as an illustrative instance to show the primary bottleneck in FHE operations. We adopt FHE encryption parameters as in [13] (e.g., $N = 2^{16}, L = 45$) for the illustrative example. As shown in Fig. 2, the horizontal roof represents the peak computation capacity (i.e., 35.6 TFLOPS) while the diagonal roof expresses the memory bandwidth (i.e., 936 GB/s) of the target hardware platform. The x-axis represents AI, which is a ratio of computation to memory access during the execution progress of CKKS operations. Overlaying the CKKS APIs on this model reveals a significant insight: all

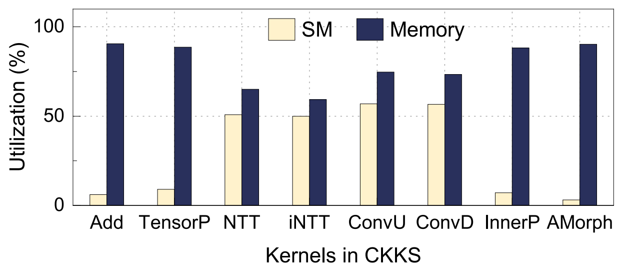


Fig. 3. SM and memory utilization during different CKKS kernels runtime.

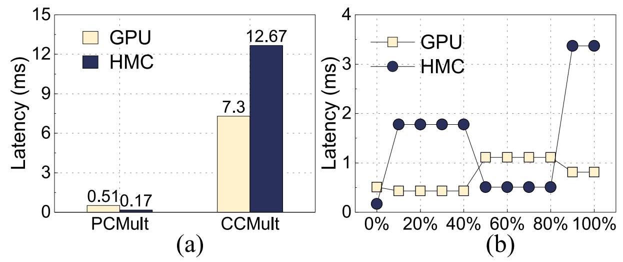


Fig. 4. Different affinity tendencies of different CKKS operations, refer to Section V for detailed settings. (a) CKKS operation. (b) Breakdown analysis.

CKKS APIs exhibit significantly low AI. In particular, these operations are situated below the diagonal roof and are far from fully exploiting the peak computational power of the hardware platform. This finding suggests that the efficiency of data movement is constraining the performance of these APIs, and simply deploying them to more powerful hardware devices may result in little boost in the overall performance.

2) *Observation 2—Different CKKS Basic Kernels Show Varying Memory Bandwidth and Computational Requirements:* Our evaluation results also reveal that although all CKKS APIs exhibit memory-intensive characteristics, the underlying basic kernels within CKKS APIs prioritize computing power and bandwidth requirements differently. We ran all CKKS kernels on an NVIDIA Geforce 3090 platform and accessed their resource utilization using the Nsight Compute [28] tool. The SM and Memory utilization results shown in Fig. 3 demonstrate while nearly all kernels exhibit memory-intensive behavior, certain kernels (e.g., *NTT, Conv*) also impose substantial demands on computation resources. This suggests that there might be kernels more suited for execution on the GPU engines. To further investigate the aforementioned issues, we choose two of the most representative CKKS APIs (i.e., *PCMult*: a) plaintext-ciphertext multiplication, comprising *TensorP* and b) *CCMult*: Ciphertext–ciphertext multiplication, comprising a series of CKKS basic kernels) and run them on GPU and HMC processing engines (for details, see Section V), respectively. Fig. 4(a) illustrates the execution latency of these two APIs on different platforms, whereas Fig. 4(b) provides the time consumption of different stages during the execution of *CCMult* on GPU and HMC sides. The results shown in Fig. 4(a) and (b) reveal two critical phenomena:

a) Different CKKS APIs exhibit distinct preferences for HMC sides. Specifically, *PCMult* tends HMC side, while *CCMult* demonstrates a preference for the GPU side.

b) Even within a single CKKS API, the various stages of its internal execution process display differing affinities for HMC side. This phenomenon primarily stems from the fact that *CCMult* encompasses all CKKS kernels outlined in Section II-A except for *AMorph*, and these kernels possess varied affinities with the HMC side.

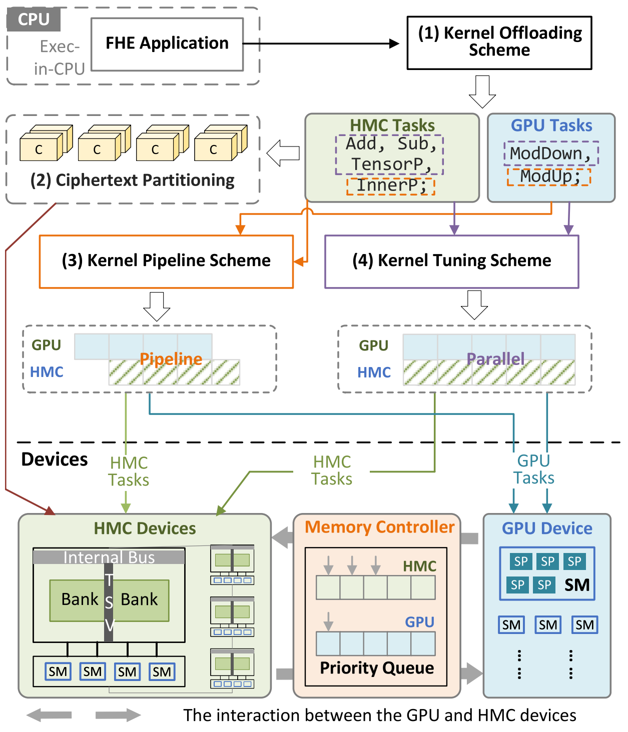


Fig. 5. Overview of HMC-FHE.

These two observations reveal the importance of bandwidth in the efficiency of CKKS operations. Additionally, they highlight another crucial point: blindly offloading FHE operations to HMC processing engines is not always advantageous. This is because certain FHE kernels still require a delicate balance between computational and bandwidth requirements. In this study, we aim to develop a kernel-level scheduling framework leveraging the GPU-HMC heterogeneous architecture to efficiently accelerate FHE operations.

III. HMC-FHE Framework

Fig. 5 provides an overview of the proposed HMC-FHE design. A typical GPU-HMC architecture is adopted as the accelerator hardware [17], [18]. This architecture integrates three distinct types of devices, each with specialized capabilities: 1) GPU, serving as the primary processing engine, focuses on computation-intensive tasks; 2) HMC, serving as a secondary processing engine and the global memory for the GPU, focuses on memory-intensive tasks; and 3) CPU, serving as an auxiliary processing device, receives applications and workloads and plays a pivotal role in overseeing task scheduling and resource management between GPU and HMC engines. To eliminate redundant data transfers between GPU and HMC devices, the HMC devices are directly connected to the GPU side via their memory links on a silicon interposer, replacing the traditional GPU global memory interface. Additionally, the interaction and data communication among these three devices is facilitated through the packet-based protocol (as mentioned in Section II-B), which is managed by the HMC devices, ensuring reliable and efficient data transfer within the system.

To accelerate an FHE application, we propose the HMC-FHE framework to automatically map and schedule the FHE operations on the GPU-HMC architecture, comprising four crucial components: 1) a fine-grained kernel offloading scheme; 2) a ciphertext partitioning scheme; 3) a kernel execution pipeline scheme; and 4) a kernel tuning scheme. These components collaborate to address three challenges as mentioned in Section I.

The fine-grained kernel offloading scheme is utilized to distribute various CKKS basic kernels to GPU or HMC engines based on their resource affinity. To enhance the parallelism of CKKS basic kernels assigned to HMC engines, we propose a ciphertext partitioning scheme aimed at achieving data parallelism across multiple HMC devices, which is achieved by mapping each ciphertext to multiple HMC devices. The kernel execution pipeline scheme leverages the loose data dependencies within the data flow graph (DFG) of those compound CKKS operations and allows their composed kernels to be executed between GPU and HMC engines in a pipeline fashion. Finally, to fully harness the capabilities of both the GPU and HMC engines, a kernel tuning module is employed to dynamically reallocate kernels between the GPU and HMC engines based on runtime status. This ensures the maximization of system parallelism.

IV. HMC-FHE Component Design

In this section, we will provide the design details of the proposed four key functional components of HMC-FHE.

A. FHE Kernel Offloading Model

Based on the analysis of FHE operations as shown in Section II-C, we propose a fine-grained FHE kernel offloading model to offload these underlying CKKS basic kernels smartly between GPU and HMC processing engines. The fundamental principle lies in the AI value of a kernel, which notably impacts its ideal processing location: kernels with lower AI tend to favor the HMC processing engine, as they require less computational power. Conversely, kernels with higher AI, signaling a greater demand for computational resources, are more effectively to be offloaded to the GPU. This concept is further illustrated in Fig. 6, showcasing a roofline model example utilizing an NVIDIA GPU with multiple HMC cubes (for details, see Section V-A). Fig. 6 show that kernels with AI values below the Critical AI (CAI, defined as the ratio of the HMC peak computation capacity to GPU bandwidth) can effectively utilize the computing resources available at the HMC processing engine. However, when the AI of a kernel

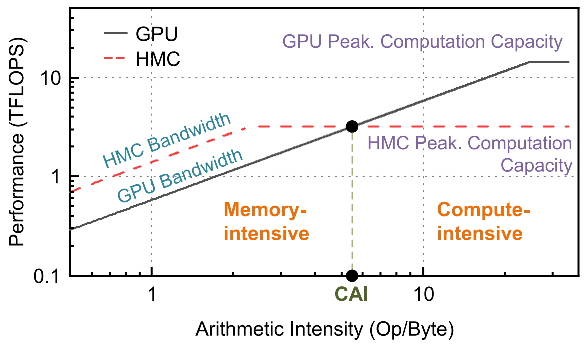


Fig. 6. Roofline model based on the GPU-HMC architecture adopted in this article.

surpasses the CAI, executing it on the GPU processing engine becomes more beneficial.

However, obtaining the AI of various CKKS kernels with different parameters in advance presents a challenge. To address this, we formulate a linear programming model for these CKKS kernels to quantify their computation and memory access ratio. This linear programming model comprises three key metrics, namely, global memory AI (MAI, the value is denoted as m), shared memory AI (SMAI, the value is denoted as s), and the ratio of shared memory instruction (SMI), count to the total instruction (TI) count (SMI: TI ratio, the value is denoted as r). MAI describes the computation-to-data access ratio when data resides in memory from an instructional perspective, while SMAI represents the same ratio when data resides in the cache. Both metrics function akin to AI, with lower values indicating reduced computational demands. Additionally, SMI:TI delineates the ratio of instructions accessing cached data to TIs, serving as a weighting mechanism to gauge the interplay between MAI, SMAI, and AI. A higher value suggests that SMAI exerts a stronger influence on AI. Note that all metrics are static and can be readily obtainable through source code parsing. Consequently, the host can seamlessly execute the kernel offloading process in real-time with negligible overhead

$$\text{Affinity} = \begin{cases} \text{HMC}, & \text{if } x_{1-r}x_m + c_1 x_r x_s < \text{CAI} \\ \text{GPU}, & \text{otherwise.} \end{cases} \quad (1)$$

Hence, the kernel offloading model is as shown in (1), where c_1 is a constant, representing the linear factors of the mapping of SMAI to CAI. x_m, x_s, x_r represent the MAI, SMAI, SMI:TI Ratio of kernel x, respectively. Accordingly, if the output of the model is less than CAI, it indicates that the current kernel is memory-intensive, implying a higher priority for its bandwidth requirement. Consequently, the kernel will be offloaded to the HMC side. Otherwise, the kernel will be offloaded to the GPU side. Table II illustrates the outcomes of various metrics and the affinity of different CKKS kernels within the context described in Section II-A.

B. Bandwidth-Aware Ciphertext Partitioning

To leverage the parallelism offered by the multiple HMCs, those basic kernels offloaded to the HMC can be processed in two ways: one approach is assigning a series of basic kernels to different HMC engines by a proper schedule mechanism. Each

TABLE II
METRICS AND THE AFFINITY RESULTS UNDER THE SETTINGS MENTIONED IN SECTION V-A. LEGEND: I) MAI AND SMAI (MEMORY TO COMPUTE RATIO $= 0 \leq L < 1 \leq M < 5 \leq H$) AND II) RATIO (SHARED INS. TO INS. $= 0 \leq L < 0.1 \leq M < 0.5 \leq H$)

Kernel	MAI	SMAI	SMI:TI Ratio	Affinity
Add	L	L	L	HMC
Sub	L	L	L	HMC
TensorP	L	L	L	HMC
iNTT	H	M	M	GPU
NTT	H	M	M	GPU
ConvU	H	M	M	GPU
ConvD	H	M	M	GPU
InnerP	L	L	L	HMC
AMorph	L	L	L	HMC

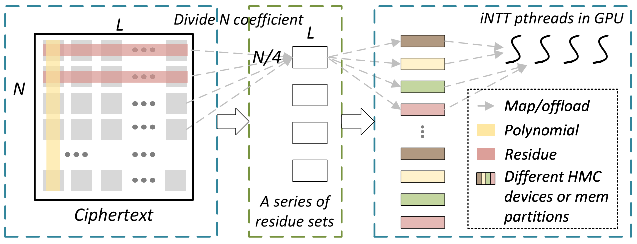


Fig. 7. Overview of Ciphertext partitioning scheme and assume that the number of HMC devices is 4. The ciphertext is composed of L polynomials and each polynomial contains N coefficients. The coefficients from the same position of the L polynomials are denoted as residue.

HMC engine manages a complete ciphertext independently, treating each HMC as an autonomous entity. Alternatively, one can accelerate a single basic kernel using multiple HMCs, with each HMC handling a portion of a single ciphertext. In this setup, all HMC engines function collectively as a unified entity. The former is heavily dependent on the DFG of applications. If there are data dependencies between basic kernels, some HMC processing engines may remain idle, leading to inefficient resource utilization. The latter allows multiple HMC engines to collaboratively execute a CKKS basic kernel, effectively guaranteeing resource utilization regardless of the DFG of applications. In this article, our focus lies on this latter method.

As mentioned in Section II-B, it was noted that the transfer latency between HMC devices exceeds three times that of local access within an HMC. Hence, the ciphertext partitioning scheme must be carefully designed to ensure its performance gain. We introduce a bandwidth-aware ciphertext partitioning (BaCP) scheme, which fulfills the following three criteria by maintaining the coherence of parallel and data access patterns within the CKKS basic kernel: 1) ensuring the parallel processing capabilities of multiple HMC engines; 2) attaining load balance across the multiple HMC engines; and 3) minimizing data transfer between HMC devices.

As shown on the left side of Fig. 7, the original dense ciphertext is organized as an $N * L$ matrix. This structure can be understood as comprising either L polynomials (each of degree N) or N residues (each of degree L). In the BaCP scheme, the ciphertext is first decomposed into four parts, with each part containing $(N/4)$ residues (assuming there are four HMC devices). Subsequently, each part is evenly distributed across different HMC devices, as illustrated on the right side of Fig. 7. For instance, considering HMC one, it manages residues indexed $1_{\text{st}}, 4_{\text{th}}, 8_{\text{th}}, \ldots, (N-3)_{\text{th}}$.

This partitioning scheme brings two advantages.

1) *Parallelism and Load Balancing:* The CKKS basic kernel, initially processed by a single HMC engine, is transformed into four parallel subkernels of equal scale and function. These four subkernels can then be concurrently processed by four HMCs, facilitating parallel execution.
2) *Decoupling of Ciphertext Structure and Access Patterns:* By decoupling the inherent structure of L polynomials, our approach enables the alignment of data access patterns and parallel patterns within CKKS basic kernels, fostering collaboration between them. Consider the scenario where the HMC side processes the *InnerP* kernel, which involves data movement across L polynomials and conducts multiply-accumulation (MAC) operations. The BaCP scheme redefines it as four smaller subkernels, each handling inputs of size $(N/4) * L$. Each subkernel then executes $(N/4)$ independent MAC operations aimed at $(N/4) * L$ residues. Within this setup, every MAC operation utilizes elements from the same residue, ensuring that all data access occurs locally within the vaults of a single HMC device.

The BaCP scheme aligns well with CKKS kernels offloaded to the HMC engines, except for *AMorph*. For kernels, such as *Add*, *Sub*, and *TensorP*, which involve element-wise operations like multiplication or addition, the BaCP scheme avoids conflicts between parallel modes and data access patterns. However, *AMorph*, due to its data access pattern being the opposite of *InnerP*, necessitates some level of data transfer between HMCs under the BaCP scheme. Nonetheless, considering the infrequent calls of *AMorph* and the relatively minimal data transfer compared to *InnerP*, the overall efficiency of the BaCP scheme is minimally impacted.

C. Kernel Execution Pipeline

While the offloading mechanism and ciphertext partitioning scheme improve the execution efficiency of CKKS kernels, the inherent data dependencies among these kernels necessitate their execution on the GPU and HMC engines in a serial manner. This introduces idle periods that reduce system parallelism and throughput. Taking the *CCMult* API as an example, which comprises four sequential CKKS operations (*TensorP* $\rightarrow$ *ModUp* $\rightarrow$ *InnerP* $\rightarrow$ *ModDown*), the kernel offloading model assigns *TensorP* and *InnerP* to the HMC engines (as indicated in Table II). Consequently, the processing engines within both the GPU and HMC engines will alternate between idle states, as depicted in Fig. 8(a). To tackle this issue, our objective is to identify loose data dependencies from interkernel within the DFG and adjust the execution sequence of certain kernels accordingly. This allows for the asynchronous execution and pipeline processing of different CKKS kernels across the GPU and HMC sides, thereby reducing the idle period of the system.

Our main focus lies on CKKS compound operations, particularly on compound kernels (e.g., *Bootstrapping*) or CKKS

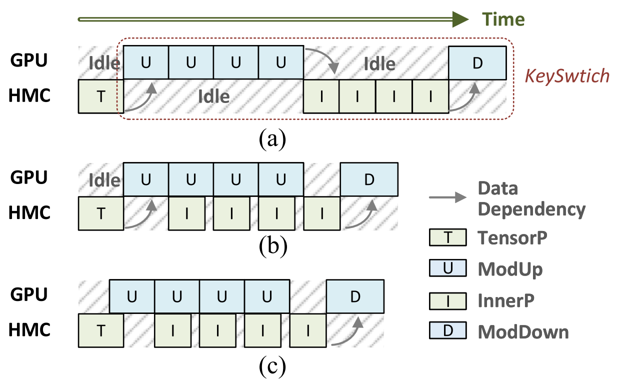


Fig. 8. Pipelined execution inside GPU-HMC architecture. *ModUP* (*ModDown*) consists of *NTT*, *INTT*, and *ConvU* (*ConvD*). The parameter *dnum* = 4. (a) Execution flow of a CCMult. (b) Loose data dependency 1. (c) Loose data dependency 2.

APIs (e.g., *CCMult, Rotate*) that incorporate the *KeySwitch* kernel. These compound operations, including *KeySwitch*, are primarily focused on for the following reasons.

1) They are the most time-consuming operations. A significant proportion of time in applications is consumed by a large number of *KeySwitch* calls, constituting over 90% of the time cost of applications [4], [29].
2) According to the kernel offloading model, these compound operations uniquely exhibit the characteristic of alternating between the GPU and HMC sides. In contrast, basic kernels like *Add* and *Sub* do not support pipelining across both sides. After a comprehensive analysis of the DFG for the mentioned operations, we identify two primary types of loose data dependencies. To illustrate these differences, we utilize the *CCMult* API as a case study:
   a) *Loose Data Dependency 1:* For the data dependency within *KeySwitch* (*ModUp* → *InnerP* → *ModDown*), we employ the General KeySwitch scheme [30] to divide the *ModUp* → *InnerP* sequence into *dnum* smaller stages. At this point, the output of each *ModUp* serves as the input for the corresponding *InnerP*, and the *dnum ModUp* → *InnerP* stages operate in complete parallelism. Consequently, we can utilize a pipelining approach to alternately execute *ModUp* and *InnerP* on the GPU and HMC engines, thereby implicitly concealing resource idleness. Subsequently, the data flow of *CCMult* transitions from the configuration depicted in Fig. 8(a) to that illustrated in Fig. 8(b).
   b) *Loose Data Dependency 2:* Another instance of loose data dependence occurs at the intersection of the *TensorP* kernel and the *KeySwitch* kernel. Specifically, *TensorP* computes four element-wise multiplications and generates three outputs, one of which serves as the input to *KeySwitch*. This implies that we can rearrange the execution sequence between *TensorP* and *KeySwitch*, allowing *TensorP* to first compute the *KeySwitch* input. Subsequently, the data dependency between *TensorP* and *KeySwitch* is eliminated, enabling their parallel execution at runtime. After restructuring the data flow, the data flow of *CCMult* transitions from the configuration depicted in Fig. 8(b) to that illustrated in Fig. 8(c), further enhancing system efficiency.

Note that the aforementioned loose data dependencies are general and can be extended to other operations that include *KeySwitch*. Therefore, the two pipeline schemes described also apply to *Rotate* and *Bootstrapping*.

D. Kernel Tuning

The kernel execution pipeline scheme primarily guarantees parallel processing capability in the system for specific kernels. However, in other scenarios, such as when the GPU engine executes *ModDown* or the HMC engine handles basic kernels like *Add* or *Sub*, the processing engines on the opposite side remain idle.

To address this issue, we propose a kernel tuning scheme aimed at optimizing resource utilization throughout the entire runtime. This scheme entails extracting subkernels from the active GPU or HMC engines and reallocating them to idle HMC or GPU engines. This ensures a more balanced and efficient utilization of resources across the architecture. It is important to note that this approach deviates from the kernel offloading model, but the enhanced utilization of system resources yields performance gains.

To ensure the efficiency of kernel execution, our kernel tuning scheme adheres to two design principles: 1) ensuring load balancing between the GPU and HMC engines and 2) minimizing the data transfer overhead between multiple HMC devices, particularly when extracting subkernels from the GPU engine and assigning them to the HMC engines. Next, we provide a detailed explanation of the kernel tuning scheme from both the GPU and HMC perspectives.

Kernels in HMC. For basic kernels offloaded to the HMC engines (e.g., *Add*, *Sub*), since they lack specific layout constraints for ciphertext elements (as discussed in Section IV-B), it is feasible to extract subkernels and reoffload them to the GPU engine. Taking the tuning process of the *Add* kernel illustrated in Fig. 9(a) as an example, its ciphertext is partitioned into two parts, with the GPU and HMC engines handling one portion each, respectively. Moreover, to ensure load balance, the ratio of data scale processed by the GPU and HMC engines adheres to

$$\frac{\text{GPU}_{\text{workload}}}{\text{HMC}_{\text{workload}}} = \frac{\text{Roofline}_G(\theta(x))}{\text{Roofline}_H(\theta(x))} \tag{2}$$

where $\theta(x) = x_{1-r}x_m + c_1x_rx_s$ (as discussed in Section IV-A), represents the AI of kernel x. $\text{Roofline}_G(\theta(x))$ and $\text{Roofline}_N(\theta(x))$ denote the available peak performance of kernel x on the roofline models of GPU and HMC, respectively. Therefore, the ratio of data scale (i.e., $\text{GPU}_{\text{workload}}/\text{HMC}_{\text{workload}}$) processed by both sides aligns with the ratio of their performance for associated kernels.

Kernels in GPU: For *ModDown*, the situation is considerably more complex. This complexity arises primarily because the *NTT* and *iNTT* kernels within *ModDown* necessitate data permutation across coefficients of each polynomial, whereas

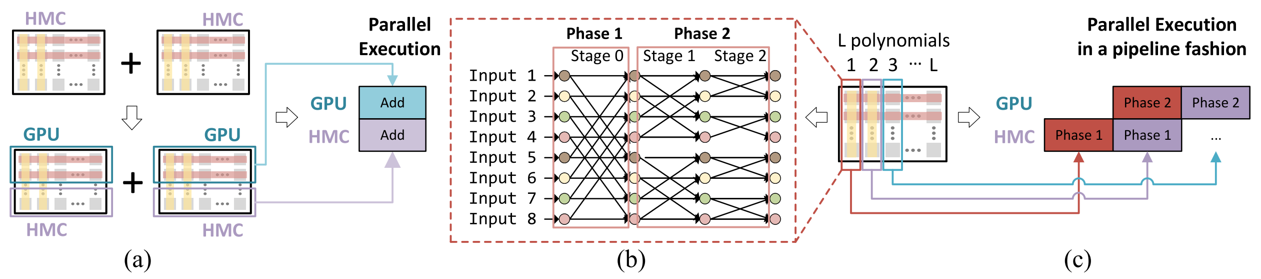


Fig. 9. Kernel tuning scheme. (a) and (c) describes the kernel tuning details between GPU and HMC engines; (b) illustrates the execution progress of an 8-input *NTT*, where different colors represent elements located in different HMC devices.

the ciphertext partitioning scheme does not allocate a complete polynomial to an HMC device. Consequently, when these kernels are offloaded to the HMC engines, it initiates extensive data exchanges between HMC devices, thereby diminishing the data movement efficiency of the system. To address this issue, the kernel tuning scheme adopts a 2-phase NTT method based on the relationship between the data access pattern within the *NTT* and the BaCP scheme.

Fig. 9(b) outlines the execution details of an 8-input *NTT*, wherein the computation process is segmented into three stages, each comprising a series of butterfly operations [31]. Suppose this kernel is reallocated to HMC engines, with the polynomial interleaved across the four HMC devices following the BaCP approach. In Stage 0, input x undergoes butterfly operations with input $x+4$ ($1 \leq x \leq 4$, stride $= 4$), where the inputs for each butterfly operation originate from the same HMC device. However, in Stage 1 or 2, the stride for the butterfly operations is 2 or 1, respectively, necessitating data exchanges between HMC devices.

We can extend this scenario to larger *NTT* inputs. For any given N, any HMC device can execute the $0 \sim \log_2(N/4) - 1$ stages of *NTT* without data transfer between HMC devices. This observation motivates us to split the *NTT* kernel into two phases and allocate phase 1 to the HMC engines. Consequently, the HMC engines can perform a portion of the *NTT* calculation under the BaCP scheme.

As depicted in Fig. 9(c), the GPU and HMC engines sequentially execute a 2-phase *NTT* for a single polynomial. For L polynomials within a ciphertext, which can be processed in parallel, the GPU and HMC effectively utilize system resources without idleness through pipeline execution. Additionally, to sustain load balance, the number of stages in phase 1 and phase 2 also conforms to (2).

Note that this approach can be readily applied to the *INTT* stage with minimal modifications. Since the *INTT* kernel is essentially the inverse of the *NTT*, the processing sequence between the GPU and HMC is simply inverted. In other words, the HMC handles phase 2 of *INTT*, while the GPU undertakes phase 1 of *INTT*.

V. Evaluation

In this section, we will provide a comprehensive overview of the experimental setup, workloads employed, evaluation results, and comparison with other implementations.

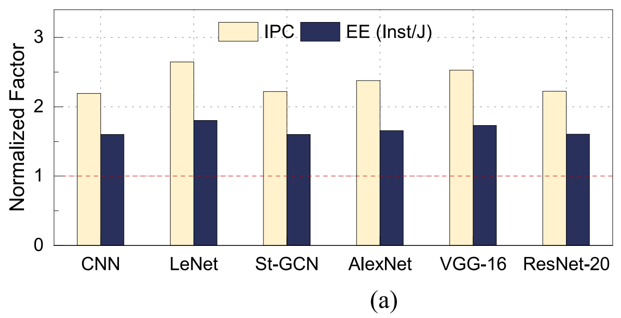
TABLE III
Platform Specifications

Baseline Configuration		
GPU platform	Total 56 SMs, 32 SIMT, GTO warp schedule, 1290MHz, L1 128 KB (per SM), Main Memory DDR6 (480 GB/s)	
GPU-HMC Configuration		
GPU side	Total 40 SMs, 32 SIMT, GTO warp schedule, 1290MHz, L1 128 KB (per SM)	
HMC side	Device	4 HMC cubes, total 32 GB capacity, 312.5MHz, 320 GB/s Internal bandwidth
	Processing logic	Total 16 SMs (4 SMs per HMC cube), 32 SIMT, 1290MHz, L1 16 KB (per SM)
	Crossbar	6-cycle all-to-all crossbar
	Serials links	10-cycle latency, including 3.2 ns for SerDes, 2 pJ/bit (1.5 pJ/bit/cycle when it is idle)
	3D DRAM timings	$t_{CK} = 1.6, t_{CAS} == 11.2, t_{RCD} = 11.2,$ $t_{RAS} = 22.4, t_{RP} = 11.2, t_{WR} = 14.4$

A. Environment Setup

Experimental Setup: We implement the prototype of HMC-FHE by integrating the latest GPGPU-Sim v4.0 [32] with the HMC simulator CasHMC [33]. Table III provides the details of the platform specifications. We configure the GPU side with 40 SM cores and each HMC processing engine with four SM cores (< 30 W, completely satisfying the thermal feasibility [34]) underneath a 3-D memory stack. We configure the platform with four HMC cubes and each cube of the HMC is embedded in a memory partition, which is directly connected to the weakened SMs located in the logic layer through an interconnection network. The external transmission latency of the memory partition is configured using the link model in BookSim [35]. In addition, we use the AccelWattch model [36] to evaluate EE under different schemes, where the energy parameters follow [37]. The diagram of the integrated architecture of GPU and HMC is as in Fig. 1.

Simulator Validation: The GPU-HMC architecture, essential to our simulations, has been extensively used in prior research, setting a reliable precedent for its use in academic studies. This guarantees that our architectural assumptions are based on tested and proven models. Our simulations are carried out using GPGPU-Sim and CasHMC, both of which are tools well-recognized within the research community for their accuracy and reliability. Moreover, previous research [32] has shown that GPGPU-Sim offers over 85% accuracy in its simulation results. This high level of precision significantly boosts the credibility of our simulation outcomes, indicating that they are a credible reflection of real-world performance.



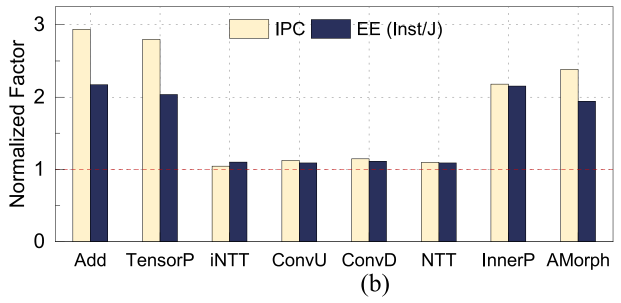


Fig. 10. (a) Throughput improvement and energy saving. (b) Performance improvement for different CKKS basic kernels.

Baseline: We compare this design with a conventional GPU architecture that has 56 SM cores as the baseline. Note that our comparison is based on a fairground of similar computational power and bandwidth resources (i.e., both of them have the same peak computing power and external interconnection bandwidth) We also evaluate this framework with more SM cores, see the details in Section V-D.

Benchmark: Six neural network models, CNN [4], LeNet [38] with dataset MNIST, St-GCN [2] with dataset NTU-RGB+D, AlexNet [38], VGG-16 [38], and ResNet-20 [16] with dataset CIFAR-10, that are typically adopted in privacy protection machine learning are used for performance evaluation. Among them, the CNN, LeNet, and AlexNet models are used for evaluating the cases without *bootstrapping*, and the other models are used for evaluating the cases with *bootstrapping*. As a result of the distinct multiplicative depth requirements (i.e., without/with *bootstrapping*) for these two types of models, we employ two separate sets of CKKS parameters denoted as $(N = 2^{16}, L = 45)$ and $(N = 2^{14}, L = 20)$, respectively. The security level $\lambda > 98$.

Implementation: For the implementation of CKKS, we adopt ckks-gpu-core [13] (the state-of-the-art GPU implementation of the CKKS scheme). We adapt parts of the implementation to enable its execution on HMC-FHE. During the runtime phase, ciphertexts along with associated constant parameters (e.g., twiddle factors in *(I)NTT*, RNS modulus generated by the CRT) involved in CKKS operations are generated in host memory and then efficiently distributed to multiple HMC devices using BaCP methodology. To effectively manage the substantial number of evaluation keys (*evks*) during model execution tasks, particularly when invoking *Bootstrap*, we devise a memory pool system. This system oversees memory allocations tailored to HMC, mitigating the risk of memory overflow. Under this framework, all *evks* are initially stored on the host side for ease of management and access. When these keys are needed for computation, the system copies an instance of Evk from the host side, formats it into a format BaCP suitable for parallel data processing, and sends it to multiple HMC devices.

B. Overall Performance

To assess the efficacy of the acceleration framework, we perform a comprehensive evaluation of end-to-end inference performance using the benchmarks mentioned above. The obtained results are subsequently compared with the baseline (i.e., GPU platform), with instructions per cycle (IPC) and EE being the primary metrics of comparison.

Fig. 10(a) provides the performance speedup and energy savings achieved across all workloads. Regarding inference latency, the HMC processing engines effectively alleviate the "memory wall" bottleneck within the CKKS kernels. Consequently, CNN, LeNet, St-GCN, AlexNet, VGG-16, and ResNet-20 exhibit performance improvements of 2.19×, 2.65×, 2.22×, 2.38×, 2.53×, 2.22× compared to the baseline, respectively. Moreover, due to the reduced power overhead of data movement, our acceleration framework decreases energy consumption across different benchmarks by 59.95%, 80.34%, 60.22%, 65.22%, 73.32%, and 60.34%, respectively. Moreover, since all workloads show similar performance gains, HMC-FHE can be well-compatible with different CKKS parameters.

To provide additional evidence of the acceleration benefits conferred by HMC-FHE across various CKKS kernels, we monitored the execution progress of the ResNet-20 models. We recorded the average execution time of different kernels and evaluated their corresponding EE from a fundamental kernel perspective. As shown in Fig. 10(b), as expected, the kernels (*Add, TensorP, InnerP, AMorph*) exhibit substantial speedup, with improvements of 2.93×, 2.79×, 2.18×, and 2.38×, respectively. Conversely, the kernels (*INTT, NTT, ConvU, ConvD*) demonstrate comparatively modest speed increases of 4.53%, 12.39%, 14.46%, and 9.68%, respectively. This is as expected. As shown in Table II, the kernels (*Add, TensorP, InnerP, AMorph*) are offloaded to HMC processing engines. Hence, they can benefit from the larger internal bandwidth, and lower latency provided by the HMC devices, resulting in significant performance improvement. On the contrary, the kernels (*INTT, NTT, ConvU, ConvD*) are performed in the GPU engine, the incompatible memory resource usage makes the acceleration effect of these kernels less obvious. Note that although various basic kernels experience different levels of speedup, their individual performance does not directly influence the overall application speedup at a global level. With the pipelined design and kernel tuning scheme, the two types of kernels running on GPU and HMC sides are executed in a parallel fashion, which to some extent conceals a significant portion of time overhead.

C. Breakdown Analysis

To evaluate the benefit of each technique toward overall performance improvement, we form combinations of different

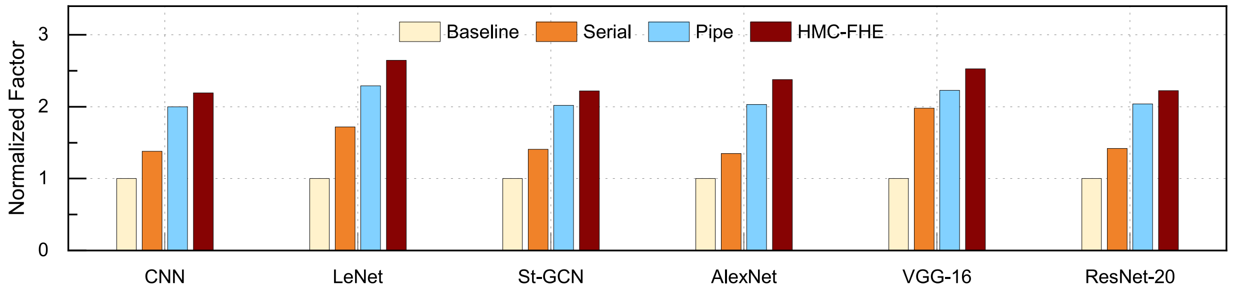
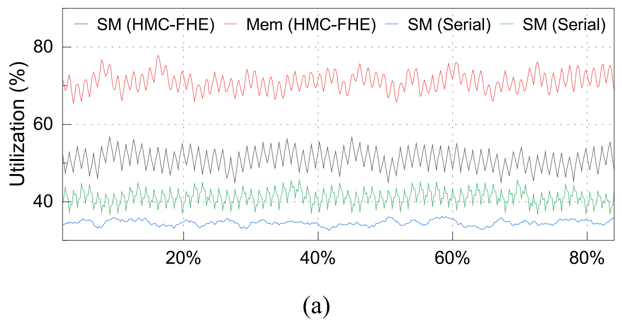


Fig. 11. Breakdown analysis of different schemes across all workloads.



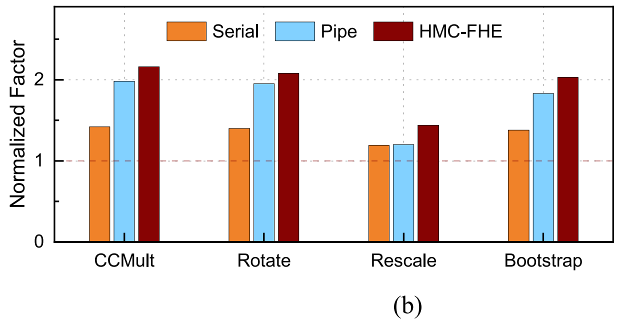


Fig. 12. (a) Resource utilization under ResNet-20. (b) Breakdown analysis of CKKS compound operations.

schemes to elucidate the advantages of these combined techniques both at the application and primitive levels. The specific combinations of schemes are outlined below.

1) *Baseline:* GPU only implementation.
2) *Serial:* Only implement the kernel offloading (Section IV-A) and the Ciphertext partition scheme (Section IV-A), with all CKKS primitives serially executed on HMC-FHE.
3) *Pipe:* Supplement the FHE operation scheduler (Section IV-C) to support pipelined execution based on Serial, but without implementing the kernel tuning scheme (Section IV-D).
4) *HMC-FHE:* Integrating all the technical schemes.

1) Application-Level: Fig. 11 presents the end-to-end inference performance of all six models with different technique combinations. By solely employing the *Serial* scheme, we observe an enhancement in the inference performance of these NN models compared to the *Baseline* by 1.38×, 1.72×, 1.41×, 1.35×, 1.98×, and 1.42×, respectively. This improvement is primarily credited to the advantages facilitated by HMC devices. Furthermore, the varied gains observed across models stem from the differing proportions of FHE operations conducted during their execution. For instance, VGG-16 is predominantly characterized by *Add* and *TensorP* kernels, with relatively fewer calls to *KeySwitch*. As a result, the *Serial* solution can fully exploit the benefits of HMC in this scenario. In comparison to the *Serial* solution, *Pipe* yielded an enhancement in inference performance by 1.44×, 1.33×, 1.43×, 1.51×, 1.13×, and 1.44×, respectively. This improvement primarily arises from the optimization of *KeySwitch* operations, which augments system parallelism while fully utilizing the architecture's overall resources. Lastly, HMC-FHE incorporates the Kernel tuning scheme atop *Pipe*, further refining the overall resource utilization of the architecture and extending parallel processing between the GPU and HMC sides to all phases of model inference. Consequently, HMC-FHE achieved a performance improvement of 1.10×, 1.16×, 1.10×, 1.17×, 1.13×, and 1.09× compared to *Pipe*.

To illustrate the variations in resource utilization across different technology combinations, using ResNet-20 as an example, Fig. 12(a) displays the average resource utilization (i.e., SM, Memory) during the execution process under both the *Serial* and *HMC-FHE* schemes. By employing kernel pipeline execution and kernel tuning, parallelism is optimized on GPU and HMC, reducing idle stalls. Consequently, the SM (weighted average of SM utilization on both GPU and HMC sides) and memory utilization in the *HMC-FHE* scheme exhibit a significant increase compared to the *Serial* scheme.

2) Compound Operations-Level: Fig. 12(b) depicts the performance improvement of all compound CKKS APIs in the aforementioned models under different technology combinations. Similar to the *application-level* scenario, the *Serial* scheme exhibits varying acceleration effects for different CKKS compound operations, yielding improvements of 1.42×, 1.40×, 1.19×, and 1.38×, respectively. Furthermore, with the additional deployment of kernel pipeline execution, the performance of these compound operations is further enhanced by 1.39 ×, 1.39 ×, 1.00 ×, and 1.33 ×, respectively. In this context, since the *Rescale* does not integrate the *KeySwitch* kernel internally, the kernel pipeline scheme is ineffective for it. Finally, with the Kernel tuning scheme, the performance sees 1.10×, 1.07×, 1.21×, and 1.11× speedup.

D. Scalability

One of the advantages of HMC-FHE is its exceptional scalability. As mentioned above, the ciphertext partitioning scheme

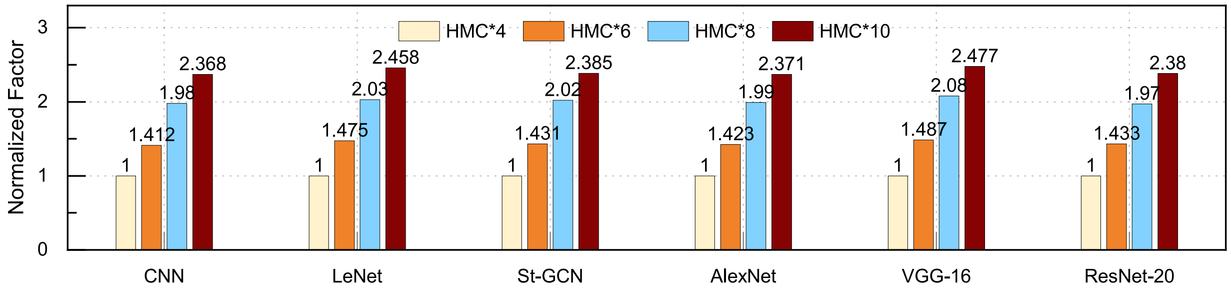


Fig. 13. Scalability evaluation for variable number of HMC devices.

effectively minimizes remote access between multiple HMC cubes while ensuring load balancing. Consequently, the HMC-FHE framework facilitates enhancing the overall parallelism of the architecture by scaling up the number of HMC devices without augmenting the complexity of interactions between HMC devices and GPU devices, or among HMC devices themselves. To quantify this characteristic, we reconfigured the number of HMC devices (e.g., 4, 6, 8, 10) in the architecture to assess the impact of changes in the number of HMCs on the overall acceleration performance of the architecture.

The results shown in Fig. 13 reveal that when there are ample computational resources on both GPU and HMC ends, HMC-FHE exhibits nearly linear performance improvement across all workloads, particularly evident in the cases of HMC*4 and HMC*8. For the remaining two cases, the performance enhancement is slightly lower than the proportion of changes in the number of HMC devices. This discrepancy arises mainly because when the number of HMCs is not a power of two, the strides within each stage of the NTT and iNTT kernels cannot consistently remain congruent with the number of HMCs, thus introducing some additional remote accesses between HMCs. In summary, the aforementioned results underscore the robust scalability of HMC-FHE.

E. Comparison With NVIDIA Tesla V100

To further demonstrate the superiority of HMC-FHE, we used it as a baseline and compared it against the actual GPU device, NVIDIA Tesla V100. We replicated the six workloads mentioned in Section V-A on the V100, with experimental setups consistent with previous tests, and the results are shown in Fig. 14. First, it was observed that across all workloads, the performance of the NVIDIA Tesla V100 consistently lagged behind HMC-FHE. In various workloads, its inference latency achieved only 0.69×, 0.57×, 0.69×, 0.64×, and 0.60× that of HMC-FHE. Additionally, the NVIDIA Tesla V100 also demonstrated disadvantages in energy consumption, with 0.59×, 0.67×, 0.60×, 0.62×, and 0.65× that of HMC-FHE.

It is noteworthy that, unlike the comparison in Section V-B between the baseline and HMC-FHE where computational power and bandwidth were aligned, the NVIDIA Tesla V100, despite having higher-peak computational power and greater bandwidth, still did not achieve an advantage in inference latency. This further validates our point: due to the different computational and memory access characteristics of various FHE kernels, merely increasing resources does not lead to

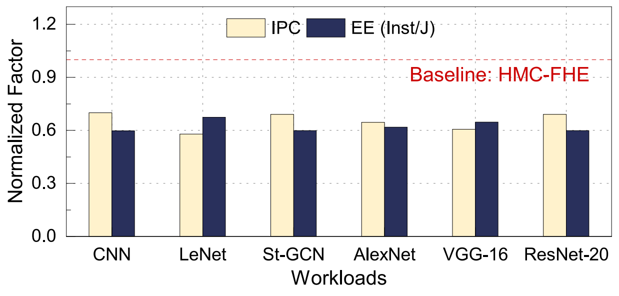


Fig. 14. Performance comparison between the HMC-FHE with four HMC devices and NVIDIA Tesla V100.

a Pareto-optimal performance solution. Appropriate kernel scheduling and resource allocation are far more crucial.

F. Comparison With Other Platforms

We also conducted comparative experiments to assess the full-system performance of HMC-FHE against other state-of-the-art accelerator prototypes, including FPGA and ASIC. To better reflect the performance of GPUs, we have set the original baseline to the NVIDIA Tesla V100. It is important to note that the computational power and bandwidth resources of the V100 are slightly greater than those of the HMC-FHE (with four HMC devices) architecture. To ensure a fair performance comparison, we standardized the CKKS encryption parameters to $N = 2^{16}, L = 45$ across the aforementioned accelerator prototypes.

1) Latency: As shown in Table IV, utilizing an HMC-FHE system with four HMC devices outperforms Poseidon [29], and F1 [5] accelerators in terms of inference latency. However, compared to BTS [7] accelerators equipped with large-capacity SRAM, the inference performance of HMC-FHE is 58% lower. Nonetheless, as previously mentioned, HMC-FHE exhibits superior scalability, and better performance can be achieved by doubling the number of HMC devices.

2) SRAM Usage: Concerning SRAM usage, Table IV illustrates that HMC-FHE consumes the least amount of SRAM. Conversely, ASIC-based acceleration designs have reached an impressive usage of 512MB of SRAM, emphasizing their significant demand for SRAM resources. This comparison underscores the advantage of HMC-FHE in resource efficiency, especially in environments with constrained SRAM resources or under heavy workloads (e.g., encryption parameters).

TABLE IV
COMPARISON OF PERFORMANCE BETWEEN HMC-FHE AND OTHER PRIOR WORKS FOR RESNET-20 INFERENCE

	Poseidon (FPGA)	F1+ (ASIC)	BTS (ASIC)	HMC-FHE (HMC*4)	HMC-FHE (HMC*8)
Time (s)	2.67	2.69	1.545	2.44	1.23
Speedup	0.91×	0.91×	1.58×	1×	2×
SRAM (MB)	8.6	256	512	9.25	14.5
Speedup	0.93×	27×	55×	1×	1.56×
Energy (W)	600	93	163	635	640
Speedup	0.95×	0.15×	0.26×	1×	1×

3) Energy Consumption: For energy consumption, the results shown in Table IV clearly demonstrate that ASIC solutions exhibit the lowest. Our proposed approach significantly improves upon the power efficiency compared to GPU deployments (as in Section V-E) and achieves energy consumption levels similar to those of FPGAs.

VI. RELATED WORK

ASIC: ASIC-based FHE accelerators, such as ARK [8], F1 [5], BTS [7], and CraterLake [16], significantly enhance the performance of FHE operations. Benefiting from the flexible resource usage and relatively more freedom in the design phase, these schemes maximize parallelism and data movement efficiency during the execution of FHE operations by customizing specialized processing units and using high-capacity SRAM.

FPGA: As a customizing-computing device, the FHE accelerator [4], [9], [10], [11], [29], [39] using FPGA adopts the idea of pipeline to optimize the execution process. HEAX [10] is a typical FPGA-accelerator in this field, introducing a pipeline-parallel architecture that maximizes parallelism from ciphertext to modular arithmetic logic. Building on this, FxHENN [4] integrates software/hardware co-design to achieve efficient resource management.

GPU: Due to its abundant parallel resources, GPU, as a general-purpose commercial device, is a promising scheme to accelerate FHE operations [12], [13], [14], [15]. TensorFHE [12] taps into the computing power of Tensor Core to accelerate the computational process of CKKS with fine-grained batch processing measures. From the perspective of compute-memory balance, 100× [13] avoids the extra memory access cost by means of operation fusion to speed up the operations.

PIM: Some works also adopt *Processing-in-Memory* to accelerate FHE operations [40], [41]. MemFHE [40] designs a configurable pipeline to accelerate FHE operations by mining the features of PIM in-situ computation and extensive parallelism. Similarly, CryptoPIM [41] optimizes memory-intensive FHE computing tasks around the excess internal bandwidth provided by the PIM architecture by building custom processing units and special memory requirements. These works focus on deploying FHE operations using only NDP devices, orthogonal to our work.

VII. CONCLUSION

In this article, we present HMC-FHE, an acceleration framework based on the heterogeneous GPU-HMC architecture to provide resource management and performance acceleration for FHE applications. HMC-FHE aims to achieve fine-grained optimization of FHE kernels with diverse features and offer optimal global task/resource scheduling to fully exploit the benefits of the GPU-HMC architecture. Various evaluation results show that compared with the SOTA GPU-based acceleration schemes, HMC-FHE achieves up to 2.65× performance improvement and reduces 1.81× energy consumption.

REFERENCES

[1] J. Lin et al, “INSPIRE: In-storage private information retrieval via protocol and architecture co-design,” in *Proc. ISCA*, 2022, pp. 102–115.
[2] R. Ran, N. Xu, W. Wang, G. Quan, J. Yin, and W. Wen, “CryptoGCN: Fast and scalable homomorphically encrypted graph convolutional network inference,” in *Proc. Adv. NeurIPS*, vol. 35, 2022, pp. 37676–37689.
[3] B. Reagen et al., “Cheetah: Optimizing and accelerating homomorphic encryption for private inference,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2021, pp. 26–39.
[4] Y. Zhu, X. Wang, L. Ju, and S. Guo, “FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference,” in *Proc. HPCA*, 2023, pp. 896–907.
[5] N. Samardzic et al., “F1: A fast and programmable accelerator for fully homomorphic encryption,” in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 238–252.
[6] N. Zhang et al., “NTTU: An area-efficient low-power NTT-uncoupled architecture for NTT-based multiplication,” *IEEE Trans. Comput.*, vol. 69, no. 4, pp. 520–533, Apr. 2020.
[7] S. Kim et al., “BTS: An accelerator for bootstrappable fully homomorphic encryption,” in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 711–725.
[8] J. Kim et al., “ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse,” in *Proc. MICRO*, 2022, pp. 1237–1254.
[9] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, “Poseidon: Practical homomorphic encryption accelerator,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2023, pp. 870–881.
[10] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, “HEAX: An architecture for computing on encrypted data,” in *Proc. ASPLOS*, 2020, pp. 1295–1309.
[11] 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 *Proc. HPCA*, 2019, pp. 387–398.
[12] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, “TensorFHE: Achieving practical computation on encrypted data using GPGPU,” in *Proc. 29th IEEE Int. Symp. High-Perform. Comput. Archit.*, 2023, pp. 922–934.
[13] 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 Trans. Cryptogr. Hardw. Embed. Syst.*, no. 4, pp. 114–148, 2021.
[14] L. de Castro et al., “Does fully homomorphic encryption need compute acceleration?” Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep. 2021/1636, 2021.
[15] W. Jung et al, “Accelerating fully homomorphic encryption through architecture-centric analysis and optimization,” *IEEE Access*, vol. 9, pp. 98772–98789, 2021.
[16] N. Samardzic et al., “CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data,” in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 173–187.
[17] N. Goswami, R. Shankar, M. Joshi, and T. Li, “Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications,” in *Proc. IISWC*, 2010, pp. 1–10.
[18] H. Jin et al., “Accelerating graph convolutional networks through a PIM-accelerated approach,” *IEEE Trans. Comput.*, vol. 72, no. 9, pp. 2628–2640, Sep. 2023.
[19] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 457–468.

[20] J. Chen, Z. Zhong, K. Sun, C. Ma, R. Mao, and Y. Wang, “Lift: Exploiting hybrid stacked memory for energy-efficient processing of graph convolutional networks,” in *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*, 2023, pp. 1–6.

[21] J. Chen et al, “GCIM: Toward efficient processing of graph convolutional networks in 3-D-stacked memory,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3579–3590, Nov. 2022.

[22] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2017, pp. 409–437.

[23] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep. 2012/144, 2012.

[24] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) fully homomorphic encryption without bootstrapping,” *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, 2014.

[25] K. Hsieh et al., “Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems,” *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 204–216, 2016.

[26] X. Zhang, S. L. Song, C. Xie, J. Wang, W. Zhang, and X. Fu, “Enabling highly efficient capsule networks processing through a PIM-based architecture design,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 542–555.

[27] S. Williams, A. Waterman, and D. A. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[28] “NVIDIA Nsight.” 2018. [Online]. Available: https://developer.nvidia.com/nsight-compute

[29] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, “Poseidon: Practical homomorphic encryption accelerator,” in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2023, pp. 870–881.

[30] K. Han and D. Ki, “Better bootstrapping for approximate homomorphic encryption,” in *Proc. Cryptogr. Track RSA Conf.*, 2020, pp. 364–390.

[31] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.

[32] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-sim: An extensible simulation framework for validated GPU modeling,” in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 473–486.

[33] D.-I. Jeon and K.-S. Chung, “CasHMC: A cycle-accurate simulator for hybrid memory cube,” *IEEE Comput. Archit. Lett.*, vol. 16, no. 1, pp. 10–13, Jan.–Jun. 2016.

[34] Y. Shen, L. Schreuders, A. Pathania, and A. D. Pimentel, “Thermal management for 3-D-stacked systems via unified core-memory power regulation,” *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 5s, pp. 1–26, Sep. 2023.

[35] N. Jiang, G. Michelogiannakis, D. Becker, and B. Towles, *BookSim 2.0 User’s Guide*, Standford Univ., Stanford, CA, USA, 2010, p. q1.

[36] V. Kandiah et al., “AccelWattch: A power modeling framework for modern GPUs,” in *Proc. 54th MICRO*, 2021, pp. 738–753.

[37] S. H. Pugsley et al., “NDC: Analyzing the impact of 3-D-stacked memory+logic devices on MapReduce workloads,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2014, pp. 190–200.

[38] Y. Cai, Q. Zhang, R. Ning, C. Xin, and H. Wu, “Hunter: HE-friendly structured pruning for efficient privacy-preserving deep learning,” in *Proc. AsiaCCS*, 2022, pp. 931–945.

[39] M. Jiang et al., “FHE-CGRA: Enable efficient acceleration of fully homomorphic encryption on CGRAs,” in presented at DAC, 2024.

[40] S. Gupta, R. Cammarota, and T. Š. Rosing, “MemFHE: End-to-end computing with fully homomorphic encryption in memory,” *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 2, pp. 1–23, 2024.

[41] H. Nejatollahi, S. Gupta, M. Imani, T. S. Rosing, R. Cammarota, and N. Dutt, “CryptoPIM: In-memory acceleration for lattice-based cryptographic hardware,” in *Proc. DAC*, 2020, pp. 1–6.