GME: GPU-based Microarchitectural Extensions to Accelerate Homomorphic Encryption

Kaustubh Shivdikar[1] Yuhui Bao[1] Rashmi Agrawal[2] Michael Shen[1] Gilbert Jonatan[3] Evelio Mora[4]
Alexander Ingare[1] Neal Livesay[1] José L. Abellán[5] John Kim[3] Ajay Joshi[2] David Kaeli[1]

[1]Northeastern University [2]Boston University [3]KAIST [4]UCAM [5]Universidad de Murcia
{shivdikar.k, bao.yu, shen.mich, ingare.a, n.livesay, d.kaeli}@northeastern.edu
{rashmi23, joshi}@bu.edu, eamora@ucam.edu, jlabellan@um.es, {gilbertjonatan, jjk12}@kaist.ac.kr

ABSTRACT

Fully Homomorphic Encryption (FHE) enables the processing of encrypted data without decrypting it. FHE has garnered significant attention over the past decade as it supports secure outsourcing of data processing to remote cloud services. Despite its promise of strong data privacy and security guarantees, FHE introduces a slowdown of up to five orders of magnitude as compared to the same computation using plaintext data. This overhead is presently a major barrier to the commercial adoption of FHE.

In this work, we leverage GPUs to accelerate FHE, capitalizing on a well-established GPU ecosystem available in the cloud. We propose GME, which combines three key microarchitectural extensions along with a compile-time optimization to the current AMD CDNA GPU architecture. First, GME integrates a lightweight on-chip compute unit (CU)-side hierarchical interconnect to retain ciphertext in cache across FHE kernels, thus eliminating redundant memory transactions. Second, to tackle compute bottlenecks, GME introduces special MOD-units that provide native custom hardware support for modular reduction operations, one of the most commonly executed sets of operations in FHE. Third, by integrating the MOD-unit with our novel pipelined 64-bit integer arithmetic cores (WMAC-units), GME further accelerates FHE workloads by 19%. Finally, we propose a Locality-Aware Block Scheduler (LABS) that exploits the temporal locality available in FHE primitive blocks. Incorporating these microarchitectural features and compiler optimizations, we create a synergistic approach achieving average speedups of 796×, 14.2×, and 2.3× over Intel Xeon CPU, NVIDIA V100 GPU, and Xilinx FPGA implementations, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Interconnection architectures**; **Very long instruction word**; Single instruction, multiple data; • **Security and privacy** → *Cryptography*; • **Networks** → *Network on chip*; • **Theory of computation** → Cryptographic primitives.

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614279

KEYWORDS

Zero-trust frameworks, Fully Homomorphic Encryption (FHE), Custom accelerators, CU-side interconnects, Modular reduction

ACM Reference Format:
Shivdikar et al. 2023. GME: GPU-based Microarchitectural Extensions to Accelerate Homomorphic Encryption. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614279

1 INTRODUCTION

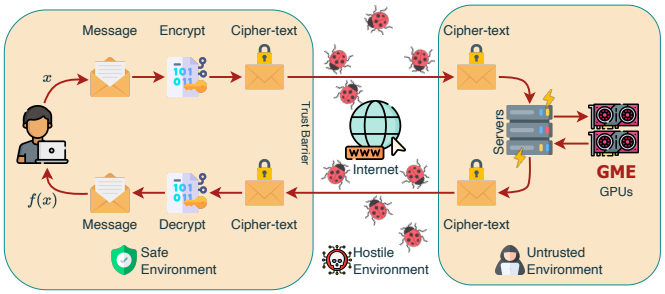


Figure 1: FHE offers a safeguard against online eavesdroppers as well as untrusted cloud services by allowing direct computation on encrypted data.

Large-scale machine learning (ML) models, such as OpenAI's GPT series and DALL-E, Google AI's BERT and T5, and Facebook's RoBERTA, have made significant advances in recent years. Unfortunately, providing public access for inference on these large-scale models leaves them susceptible to zero-day exploits [38, 71]. These exploits expose the user data as well as the ML models to hackers for potential reverse engineering [38], a concerning prospect as these models are highly valued assets for their respective companies. For example, a recent security vulnerability in the Redis client library resulted in a data breach on ChatGPT [60], which is currently regarded as one of the leading machine learning research platforms.

In the past decade, Fully Homomorphic Encryption (FHE) has emerged as the "holy grail" of data privacy. Using FHE, one can perform operations on encrypted data without decrypting it first (see Figure 1). FHE adopters can offload their encrypted private data to third-party cloud service providers while preserving end-to-end privacy. Specifically, the *secret key* used for encryption by users

is never disclosed to the cloud providers, thus facilitating privacy-preserving ML training and inference in an untrusted cloud setting (whether self-hosted or utilizing public cloud services) [77, 83, 87].

During its early stages, homomorphic encryption was limited by the number and types of computations, rendering it viable solely for *shallow circuits* [30]. In these circuits, the error would propagate and increase with each addition or multiplication operation, ultimately leading to decryption errors. Following Gentry's groundbreaking work [30], this important limitation was resolved by using *bootstrapping* [19], resulting in FHE computations that permit an unlimited number of operations. Although FHE offers significant benefits in terms of privacy preservation, it faces the challenge of being extremely slow (especially the bootstrapping operation), with performance up to five orders of magnitude slower than plaintext computing [42].

Prior studies have tried to accelerate FHE kernels by developing CPU extensions [15, 31, 42, 55], GPU libraries [4, 54, 61, 76], FPGA implementations [1, 66, 88], and custom accelerators [33, 45, 67]. CPU-based solutions inherently face limitations due to their limited compute throughput [17], while FPGA-based solutions are constrained by their limited operating frequency and resources available on the FPGA board. ASIC-based solutions provide the most acceleration [29], but they cannot be easily adapted to future algorithmic changes and can be fairly expensive to use in practice. Additionally, as the number of diverse domain-specific custom accelerators grows rapidly, it becomes increasingly difficult to create high-quality software libraries, compilers, drivers, and simulation tools for each accelerator in a timely manner, posing a challenge in terms of time-to-market. Therefore, while previous work has accelerated FHE workloads, they often fall short in terms of cost-effectiveness or lack the necessary infrastructure to support large-scale deployment.

Rather than developing domain-specific custom accelerators, our work focuses on enhancing the microarchitecture of GPUs that are currently deployed in the cloud and can be easily upgraded. This leads to a practical solution as we can readily exploit the cloud ecosystem that is built around GPUs. On the upside, GPUs offer a large number of vector processing units, so they are a good match to capitalize on the inherent parallelism associated with FHE workloads. However, FHE ciphertexts are large (dozens of MB), require a massive number of integer arithmetic operations, and exhibit varying stride memory access patterns. This imposes a true challenge for existing GPU architectures since GPUs have been historically designed to excel at executing thousands of threads in parallel (e.g., batched machine-learning workloads) featuring uniform memory access patterns and rich floating-point computations.

To bridge the wide performance gap between operating on encrypted data using FHE and operating on plaintext data in GPUs, we propose several microarchitectural features to extend the latest AMD CDNA GPU architecture. Specifically, our efforts are focused on improving the performance of the Residue Number System (RNS) version of the CKKS FHE scheme, as it naturally supports numerous privacy-preserving applications. Similar to results found in earlier studies [24], our benchmarking of CKKS FHE kernels indicates they are significantly bottlenecked by the limited main memory bandwidth. This is because current GPUs suffer from excessive redundant memory accesses when executing FHE-based workloads.

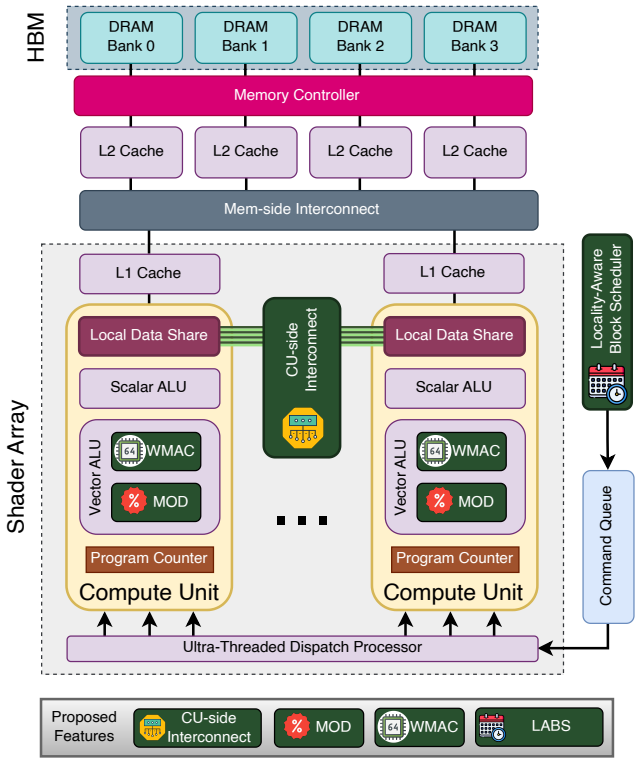


Figure 2: The four key contributions of our work (indicated in green) evaluated within the context of an AMD CDNA GPU architecture.

Present GPUs are ill-equipped to deal with varying stride FHE memory access patterns. According to our experiments, this can lead to a very high degree of compute unit stalls and is a primary cause of the huge performance slowdown in FHE computations on GPU-based systems.

To address these challenges, we propose GME, a hardware-software co-design specifically tailored to provide efficient FHE execution on the AMD CDNA GPU architecture (illustrated in Figure 2). First, we present *CU-side interconnects* that allow ciphertext to be retained within the on-chip caches, thus eliminating redundant memory transactions in the FHE kernels. Next, we optimize the most commonly executed operations present in FHE workloads (i.e., the modular reduction operations) and propose novel *MOD-units*. To complement our *MOD-units*, we introduce *WMAC-units* that natively perform 64-bit integer operations, preventing the throttling of the existing 32-bit arithmetic GPU pipelines. Finally, in order to fully benefit from the optimizations applied to FHE kernels, we develop a Locality-Aware Block Scheduler (LABS) that enhances the temporal locality of data. LABS is able to retain on-chip cache data across FHE blocks, utilizing block computation graphs for assistance.

To faithfully implement and evaluate GME, we employ NaviSim [11], a cycle-accurate GPU architecture simulator that accurately models the CDNA ISA [6]. To further extend our research to capture inter-kernel optimizations, we extend the implementation of NaviSim with a block-level directed acyclic compute graph

simulator called BlockSim. In addition, we conduct ablation studies on our microarchitectural feature implementations, enabling us to isolate each microarchitectural component and evaluate its distinct influence on the entire FHE workload.

Our contributions include:

(1) *Simulator Infrastructure:* We introduce BlockSim, which, to the best of our knowledge, is among the first efforts to develop a simulator extension for investigating FHE microarchitecture on GPUs.
(2) *CU-side interconnect (**cNoC**):* We propose an on-chip network that interconnects on-chip memory, enabling the exploitation of the large on-chip memory capacity and support for the all-to-all communication pattern commonly found in FHE workloads.
(3) *GPU Microarchitecture:* We propose microarchitectural enhancements for GPUs, including ISA extensions, modular reduction operation microarchitecture, and a wide arithmetic pipeline to deliver high throughput for FHE workloads.
(4) *Locality-Aware Block Scheduler*: Utilizing the CU-side interconnect (**cNoC**), we propose a graph-based block scheduler designed to improve the temporal locality of data shared across FHE primitives.

Our proposed improvements result in an average speedup of 14.6× over the prior state-of-the-art GPU implementation [41] for HE-LR and ResNet-20 FHE workloads. Our optimizations collectively reduce redundant computation by 38%, decreasing the memory pressure on DRAM. Although the proposed optimizations can be adapted for other architectures (with minor modifications), our work primarily concentrates on AMD's CDNA microarchitecture MI100 GPU.

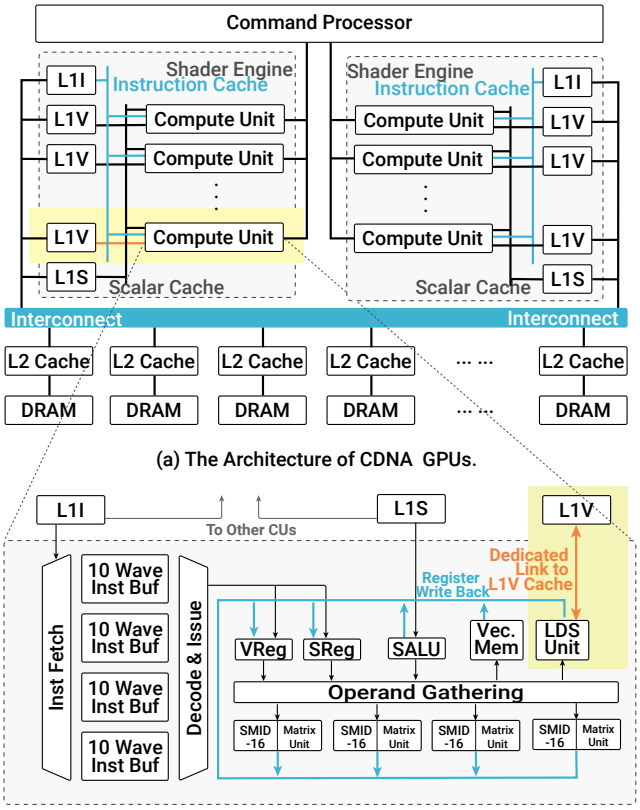
2 BACKGROUND

In this section, we briefly describe the AMD CDNA architecture and background of the CKKS FHE scheme.

2.1 AMD CDNA Architecture

To meet the growing computation requirements of high-performance computing (HPC) and machine learning (ML) workloads, AMD introduced a new family of CDNA GPU architectures [8] that are used in AMD's Instinct line of accelerators. The CDNA architecture (see Figure 3) adopts a highly modular design that incorporates a Command Processor (CP), Shader Engines (including Compute Units and L1 caches), an interconnect connecting the core-side L1 caches to the memory-side L2 caches and DRAM. The CP receives requests from the driver on the CPU, including memory copying and kernel launch requests. The CP sends memory copying requests to the Direct Memory Access (DMA), which handles the transfer of data between the GPU and system memory. The CP is also responsible for breaking kernels down into work-groups and wavefronts, sending these compute tasks to Asynchronous Compute Engines (ACE), which manage the dispatch of work-groups and wavefronts on the Compute Units (CUs).

The CDNA architecture employs the CU design from the earlier GCN architecture but enhances it with new Matrix Core Engines. A CU (see Figure 3) is responsible for instruction execution and data processing. Each CU is composed of a scheduler that can fetch



(b) The Architecture of a CDNA Compute Unit.

Figure 3: Architecture diagram showing the limitations of AMD GPU memory hierarchy. Each compute unit has a dedicated L1V cache and an LDS unit that cannot be shared with neighboring compute units.

and issue instructions for up to 40 wavefronts. Different types of instructions are issued to different execution units, including a branch unit, scalar processing units, and vector processing units. The scalar processing units are responsible for executing instructions that manipulate data shared by work-items in a wavefront. The vector processing units include a vector memory unit, four Single-Instruction Multiple-Data (SIMD) units, and a matrix core engine. Each SIMD unit is equipped with 16 single-precision Arithmetic Logic Units (ALUs), which are optimized for FP32 operations. The matrix core engine handles multiply-accumulate operations, supporting various datatypes (like 8-bit integers (INT8), 16-bit half-precision FP (FP16), 16-bit Brain FP (bf16), and 32-bit single-precision FP32). We cannot leverage these engines for FHE, as they work with INT8 operands that are not well-suited for FHE computations [78] (FHE workloads benefit from INT64 arithmetic pipelines). Each CU has a 64 KB memory space called the Local Data Share (LDS), which enables low-latency communication between work-items within a work-group. LDS is analogous to shared memory in CUDA. This memory is configured with 32 banks to achieve low latency and high bandwidth access. LDS facilitates effective data sharing among work-items and acts as a software cache to minimize global memory

accesses. However, a significant limitation of LDS is that CUs can only access its local LDS, and directly accessing remote LDS is not possible.

The CDNA architecture has a two-level cache hierarchy. Each CU has a dedicated L1 vector cache. CUs in a Shader Engine (typically 15 CUs) share an L1 scalar cache and an L1 instruction cache. The second level of cache is composed of memory-side L2 caches. Each L2 cache interfaces to a DRAM controller (typically implemented in HBM or GDDR technology). The L2 caches and the DRAM controllers are banked, allowing them to service a part of the address space.

2.2 CKKS FHE Scheme

In this paper, we focus on the CKKS FHE scheme, as it can support a wide range of privacy-preserving applications by allowing operations on floating-point data. We list the parameters that define the CKKS FHE scheme in Table 1 and the corresponding values of key parameters in Table 3. The main parameters —i.e., N and Q— define the size of the ciphertext and also govern the size of the working data set that is required to be present in the on-chip memory. The ciphertext consists of a pair of elements in the polynomial ring $R_Q = \mathbb{Z}_Q[x]/(x^N + 1)$. Each element of this ring is a polynomial $\sum_{i=0}^{N-1} a_i x^i$ with "degree-bound" $N - 1$ and coefficients a_i in $\mathbb{Z}_Q$. For a message $\mathbf{m} \in \mathbb{C}^n$, we denote its encryption as $[\![\mathbf{m}]\!] = (\mathrm{A}_\mathbf{m}, \mathrm{B}_\mathbf{m})$ where $\mathrm{A}_\mathbf{m}$ and $\mathrm{B}_\mathbf{m}$ are the two polynomials that comprise the ciphertext.

For 128-bit security, typical values of N range from 2^{16} to 2^{17} and $\log Q$ values range from 1700 to 2200 bits for practical purposes. These large sizes of N and $\log Q$ are required to maintain the security of the underlying Ring-Learning with Errors assumption [57]. However, there are no commercially available compute systems that have hundred-bit wide or thousand-bit wide ALUs, which are necessary to process these large coefficients. A common approach for implementing the CKKS scheme on hardware with a much smaller word length is to choose Q to be a product of distinct word-sized primes $q_1, \ldots, q_\ell$. Then $\mathbb{Z}_Q$ can be identified with the "product ring" $\prod_{i=1}^{l} \mathbb{Z}_{q_i}$ via the Chinese Remainder Theorem [79]. In practice, this means that the elements of $\mathbb{Z}_Q$ can be represented as an ℓ-tuple $(x_1, \ldots, x_\ell)$ where $x_i \in \mathbb{Z}_{q_i}$ for each i. This representation of elements in $\mathbb{Z}_Q$ is referred to as the *Residue Number System* (RNS) and is commonly referred to as the limbs of the ciphertext.

In this work, as shown in Table 3, we choose $N = 2^{16}$ and $\log Q = 1728$, meaning that our ciphertext size will be 28.3 MB, where each polynomial in the ciphertext is ~14 MB. After RNS decomposition on these polynomials using a word length of 54 bits, we get 32 limbs in each polynomial, where each limb is ~ 0.44 MB large. The last level cache and the LDS in the AMD MI100 are 8 MB and 7.5 MB, respectively. Thus we cannot accommodate even a single ciphertext in the on-chip memory. At most, we can fit ~18 limbs of a ciphertext polynomial, and as a result, we will have to perform frequent accesses to the main memory to operate on a single ciphertext. In addition, the large value of N implies that we need to operate on 2^{16} coefficients for any given homomorphic operation. The AMD MI100 GPU includes 120 CUs with 4 SIMD units each. Each SIMD unit can execute 16 threads in parallel. Therefore, a total of 7680 operations (scalar additions/multiplications) can be performed in parallel. However, we need to schedule the operations on 2^{16} coefficients in over eight batches (2^{16} / 7680), adding to the complexity of scheduling operations.

Table 1: CKKS Parameters and descriptions

Param	Description
N	Polynomial degree-bound
n	Length of the message. $n \leq \frac{N}{2}$
Q	Polynomial modulus
L	Maximum number of limbs in a ciphertext
C	The set $\{q_0, q_1, \ldots, q_L\}$ of prime factors of Q
ℓ	Number of limbs, number of factors in Q;
dnum	Number of digits in the switching key
α	Number of limbs that comprise a single digit in the key-switching decomposition $\alpha = \lceil \frac{L+1}{\text{dnum}} \rceil$
P	Product of extension limbs added for raised modulus. Total extension limbs = $\alpha + 1$
fftIter	Multiplicative depth of bootstrapping linear transform
Δ	Scale multiplied during encryption
$\mathbf{m}$	A message vector of n slots
$[\![\mathbf{m}]\!]$	Ciphertext encrypting a message
$\mathrm{A}_\mathbf{m}$	A randomly sampled polynomial from message $\mathbf{m}$
P	Encrypted message as a polynomial
P_m	Polynomial encrypting message m
$[P]_{q_i}$	q_i-limb of P
evk	Evaluation key
$\mathbf{evk}_{rot}^{(r)}$	Evaluation key for *HE-Rotate* block with (r) rotations

We list all the building blocks in the CKKS scheme in Table 2. All of the operations that form the building blocks of the CKKS scheme reduce to 64 bit-wide scalar modular additions and scalar modular multiplications. The commercially available GPU architectures do not implement these wide modular arithmetic operations directly, but can emulate them via multiple arithmetic instructions, which significantly increases the amount of compute required for these operations. Therefore, providing native modular arithmetic units is critical to accelerating FHE computation. To perform modular addition over operands that are already reduced, we use the standard approach of conditional subtraction if the addition overflows the modulus. For generic modular multiplications, we use the modified Barrett reduction technique [76].

The ScalarAdd and ScalarMult are the two most basic building blocks that add and multiply a scalar constant to a ciphertext. PolyAdd and PolyMult add and multiply a plaintext polynomial to a ciphertext. We define separate ScalarAdd and ScalarMult operations (in addition to PolyAdd and PolyMult) because the scalar constant values can be fetched directly from the register file that can help save expensive main memory accesses. Note that the PolyMult is followed by an HERescale operation to restore the scale of a ciphertext to Δ from scale Δ^2. The CKKS supports floating-point messages, so all encoded messages must include a scaling factor Δ. This scaling factor is typically the size of one of the limbs of the ciphertext. When multiplying messages together, this scaling factor grows as well. The scaling factor must be shrunk down in order to avoid overflowing the ciphertext coefficient modulus.

Table 2: HE building blocks using CKKS

Block	Computation	Description
ScalarAdd($[\![\mathbf{m}]\!], c$)	$[\![\mathbf{m}+\mathbf{c}]\!] = (\mathsf{B}_\mathbf{m}+\mathbf{c}, \mathsf{A}_\mathbf{m})$	Add a scalar c to a ciphertext where, $\mathbf{c}$ is a length-N vector with every element c
ScalarMult($[\![\mathbf{m}]\!], c$)	$[\![\mathbf{m}\cdot\mathbf{c}]\!] = (\mathsf{B}_\mathbf{m}\cdot\mathbf{c}, \mathsf{A}_\mathbf{m}\cdot\mathbf{c})$	Multiply a scalar by a ciphertext
PolyAdd($[\![\mathbf{m}]\!], \mathsf{P}_{\mathbf{m}'}$)	$[\![\mathbf{m}+\mathbf{m}']\!] = (\mathsf{B}_\mathbf{m}+\mathsf{P}_{\mathbf{m}'}, \mathsf{A}_\mathbf{m})$	Add an unencrypted polynomial to a ciphertext
PolyMult($[\![\mathbf{m}]\!], \mathsf{P}_{\mathbf{m}'}$)	$[\![\mathbf{m}\cdot\mathbf{m}']\!] = (\mathsf{B}_\mathbf{m}*\mathsf{P}_{\mathbf{m}'}, \mathsf{A}_\mathbf{m}*\mathsf{P}_{\mathbf{m}'})$	Multiplying an unencrypted polynomial with a ciphertext
HEAdd($[\![\mathbf{m}]\!], [\![\mathbf{m}']\!]$)	$[\![\mathbf{m}+\mathbf{m}']\!] = (\mathsf{B}_\mathbf{m}+\mathsf{B}_{\mathbf{m}'}, \mathsf{A}_\mathbf{m}+\mathsf{A}_{\mathbf{m}'})$	Add two ciphertexts
HEMult($[\![\mathbf{m}]\!], [\![\mathbf{m}']\!], \mathbf{evk}_{\text{mult}}$)	$[\![\mathbf{m}\cdot\mathbf{m}']\!] = \text{KeySwitch}(\mathsf{A}_\mathbf{m}*\mathsf{A}_{\mathbf{m}'}, \mathbf{evk}_{\text{mult}})+$ $(\mathsf{B}_\mathbf{m}*\mathsf{B}_{\mathbf{m}'}, \mathsf{A}_\mathbf{m}*\mathsf{B}_{\mathbf{m}'}+\mathsf{A}_{\mathbf{m}'}*\mathsf{B}_\mathbf{m})$	Multiply two ciphertexts
HERotate($[\![\mathbf{m}]\!], r, \mathbf{evk}^{(\mathbf{r})}_{\text{rot}}$)	$[\![\mathbf{m}\ll r]\!] = \text{KeySwitch}(\psi_r(\mathsf{A}_\mathbf{m}), \mathbf{evk}^{(\mathbf{r})}_{\text{rot}})+$ $(\psi_r(\mathsf{B}_\mathbf{m}), \mathbf{0})$	Circular rotate elements left by r slots ψ_r is an automorphism performed
HERescale($[\![\mathbf{m}]\!]$)	$[\![\Delta^{-1}\cdot\mathbf{m}]\!] = (\Delta^{-1}\mathsf{B}_\mathbf{m}, \Delta^{-1}\mathsf{A}_\mathbf{m})$	Restore the scale of a ciphertext from scale Δ^2 back to Δ

In order to enable fast polynomial multiplication, by default, we represent polynomials as a series of N evaluations at fixed roots of unity. This allows polynomial multiplication to occur in $O(N)$ time instead of $O(N^2)$ time. We refer to this polynomial representation as the *evaluation representation*. There are certain sub-operations within the building blocks, defined in Table 2, that operate over the polynomial's *coefficient representation*, which is simply a vector of its coefficients. Moving between the two polynomial representations requires a number-theoretic transform (NTT) or inverse NTT, which is the finite field version of the fast Fourier transform (FFT). We incorporate a merged-NTT algorithmic optimization [65], improving spatial locality for twiddle factors as they are read sequentially.

The HEAdd operation is straightforward and adds the corresponding polynomials within the two ciphertexts. However, the HEMult and HERotate operations are computationally expensive as they perform a KeySwitch operation after the multiplication and automorph operations, respectively. In both the HEMult and HERotate implementations, there is an intermediate ciphertext with a decryption key that differs from the decryption key of the input ciphertexts. In order to change this new decryption key back to the original decryption key, we perform a key switch operation. This operation takes in a switching key (either $\mathbf{evk}_{\text{mult}}$ or $\mathbf{evk}^{(\mathbf{r})}_{\text{rot}}$) and a ciphertext $[\![\mathbf{m}]\!]_s$ that is decryptable under a secret key s. The output of the key switch operation is a ciphertext $[\![\mathbf{m}]\!]_{s'}$ that encrypts the same message but is decryptable under a different key s'.

To incur minimal noise growth during the key switch operation, the key switch operation requires that we split the polynomial into dnum digits, then raise the modulus before multiplying with the switching key followed by a modulus down operation. The modulus raise and down operations operate on the coefficient representation of the polynomial, requiring us to perform expensive NTT and iNTT conversions. Moreover, the switching keys are the same size as the ciphertext itself, requiring us to fetch ~112 MB of data to multiply the switching keys with the ciphertext. Thus, the key switching operation not only adds to the bulk of the compute through hundreds of NTT and iNTT operations, but also leads to memory bandwidth bottlenecks. Finally, there exists an operation known as bootstrapping [30] that needs to be performed frequently to de-noise the ciphertext. This bootstrapping operation is a sequence of the basic building blocks in the CKKS scheme, meaning that it suffers from the same compute and memory bottlenecks that exist in these building blocks, making it one of the most expensive operations.

Table 3: Practical parameters for our FHE operations.

$\log(q)$	N	$\log Q$	L	L_{boot}	dnum	fftIter	λ
54	2^{16}	1728	23	17	3	4	128

3 GME ARCHITECTURE

The current issue with GPUs while implementing FHE workloads is the significant disproportion in the usage of various hardware resources present on the GPUs. As a result, specific resources such as CUs experience underutilization, while others, like HBM and on-chip caches, pose as significant bottlenecks. In this paper, we propose to re-architect the current GPU microarchitecture and also introduce novel microarchitectural extensions that enable optimal utilization of GPU resources so as to maximize the performance of the FHE workloads running on the GPU. We propose **GME**, a robust set of microarchitectural features targeting AMD's CDNA architecture, unlocking the full potential of the GPU to accelerate FHE workloads over 14.2× as compared to the previous comparable accelerators [41].

In our work, we pinpoint critical bottlenecks encountered during FHE workload execution and address them progressively using four microarchitectural feature extensions. Our on-chip CU-side hierarchical network (**cNoC**) and the Locality Aware Block Scheduler

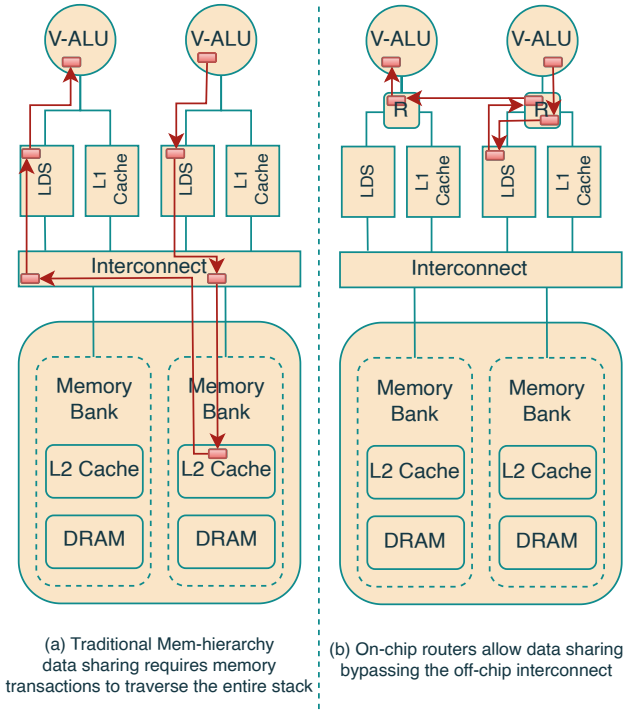


Figure 4: Inter-CU communication: Traditional vs proposed communication with on-chip network

(**LABS**) contribute to minimizing the DRAM bandwidth bottleneck. Simultaneously, our implementation of native modular reduction (**MOD**) and wider multiply-accumulate units (**WMAC**) features improve the math pipeline throughput, ensuring a streamlined data flow with evenly distributed resource utilization. The list and impact of our contributions can be visualized in Figure 2.

3.1 cNoC: CU-side interconnect

Modern GPUs have a network-on-chip that interconnects the cores (in the case of AMD GPUs, compute units) together with the memory partitions or memory banks. In this work, we propose a new type of on-chip interconnect that we refer to as a *CU-side network-on-chip (**cNoC**) that interconnects the CUs together* – in particular, all the CU's LDS are interconnected together with (**cNoC**) to enable a "global" LDS that can be shared between the CUs. By exploiting the (**cNoC**), the dedicated on-chip memory can be shared between cores, thus minimizing main memory accesses. Within our research, we specifically adapted the (**cNoC**) to serve our FHE workload. By leveraging the "global" LDS facilitated by the (**cNoC**), FHE ciphertexts that reside in the LDS can be effortlessly shared among neighboring compute units. This not only streamlines operations but, more crucially, eliminates the need to store data in the main memory and subsequently reload it for sharing across cores. This approach significantly reduces latency, as direct core-to-core sharing via the (**cNoC**) bypasses the often time-consuming main memory accesses.

We also provide synchronization barriers of varying granularity to mitigate race conditions. Since the LDS is user-controlled, our approach does not incur the overhead associated with cache coherence and avoids redundant cache invalidations, but comes with some extra programmer effort. By implementing a global address space (GAS) in our GPU, we establish data sharing and form a unified GAS by combining all LDSs. The virtual address space is then mapped onto this unified GAS, with translation using a hash of the lower address bits.

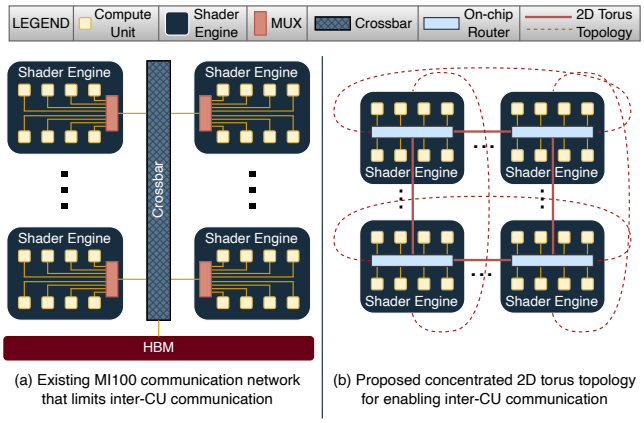


Figure 5: Proposed hierarchical on-chip network featuring a concentrated 2D torus topology

Current GPUs are designed hierarchically – e.g., MI100 GPU comprises numerous compute units, with 8 of them combined to form a *Shader Engine* (seen in Figure 5). The proposed (**cNoC**) takes advantage of this hierarchy, utilizing a hierarchical on-chip network (illustrated in Figure 5) that features a single router for each *Shader Engine*, connecting the eight compute units that make up a *Shader Engine*. The MI100 GPU houses 15 *Shader Engines*, resulting in a total of 120 compute units. The routers are arranged in a 3×5 2D grid and interconnected through a torus topology. While this *concentrated-torus* topology [10, 39] can increase network complexity, it reduces the number of required routers (from 120 to 15), thereby minimizing the chip area needed for the network. In a concentrated-torus topology, all routers have the same degree (number of ports), creating an edge-symmetric topology that is well-suited for the all-to-all communication patterns of FHE workloads.

Figure 4(a) illustrates the conventional approach of data sharing, where memory transactions must traverse through the full memory hierarchy to share data between neighboring LDS. In contrast, our proposed CU-side interconnect, presented in Figure 4(b), incorporates on-chip routers that circumvent off-chip interconnects, improving data reuse. This results in a decrease of redundant memory operations by 38%, effectively supporting the all-to-all communication pattern commonly seen in FHE workloads.

3.2 Enhancing the Vector ALU

Native modular reduction extension: (MOD) The existing GPU arithmetic pipeline is highly optimized for data manipulation operations like *multiply*, *add*, *bit-shift*, and *compare*. A wavefront executing any of these instructions takes 4 clock cycles in a lock-step manner in the SIMD units. In a single wavefront consisting of 64 threads, 16 threads are executed concurrently on the SIMD units during each clock cycle. Conversely, operations like *divide*

Table 4: Cycle counts for 64-bit modulus instructions comparing MOD and WMAC features

μ-arch. Feature	mod-red (cycles)*	mod-add (cycles)*	mod-mul (cycles)*
Vanilla MI100†	46	62	63
MOD$^{\Delta}$	26	18	38
MOD+WMAC	17	7	23

† Refers to the unmodified CDNA architecture of MI100 GPUs.
*Cycle count is averaged over 10,000 modulus instructions computed on cached data (using LDS cache) and rounded to the nearest integer.
$^{\Delta}$Modular operation is computed with various compile-time prime constants as modulus incorporating compiler optimizations into the performance.

and *modulus* are emulated using a series of native instructions, resulting in considerably slower performance compared to their native counterparts.

As stated in Section 2.2, the modular reduction operation, used for determining the remainder of a division, is performed after each addition and multiplication. As a result, optimizing modular reduction is crucial for speeding up FHE workloads. At present, the MI100 GPU executes a modular operation through a sequence of addition, multiplication, bit shift, and conditional operations, drawing on the conventional Barrett's reduction algorithm [48]. This operation currently takes a considerable amount of time, with the `mod-red` operation requiring an average of 46 cycles for execution on the MI100 GPU. In our study, we suggest enhancing the Vector ALU pipeline within the CDNA architecture to natively support modular reduction, which brings it down to an average of 17 cycles for each `mod-red` instruction. We augment the CDNA instruction set architecture (ISA) with a collection of vector instructions designed to perform modular reduction operations natively after addition or multiplication operations. The new native modular instructions proposed include:

- Native modular reduction:
  `mod-red <v0,s0>` | $\mathbf{V}_0 = \mathbf{V}_0 \bmod s_0$
- Native modular addition:
  `mod-add <v0,v1,s0>` | $\mathbf{V}_0 = (\mathbf{V}_0 + \mathbf{V}_1) \bmod s_0$
- Native modular multiplication:
  `mod-mult <v0,v1,s0>` | $\mathbf{V}_0 = (\mathbf{V}_0 \times \mathbf{V}_1) \bmod s_0$

Modular reduction involves several comparison operations, resulting in branch divergence in GPUs. Our implementation is derived from an improved Barrett's reduction algorithm [76]. This approach minimizes the number of comparison operations to one per modular reduction operation, significantly reducing the number of branch instructions and enhancing compute utilization.

Wider multiply-accumulate units (WMAC): In the CKKS FHE scheme, we can choose to perform operations on 32, 64, or 128-bit wide RNS limbs for a ciphertext. This limb bit width governs the operand size for the vector ALUs, impacting the number of modular addition and multiplication operations required. Moreover, there is an algorithmic-level performance versus precision tradeoff to consider when deciding on the bit width. If we opt for 32-bit wide RNS limbs, we will have numerous limbs to work with, increasing the available levels [2] while simultaneously reducing the achievable precision for an application. Conversely, if we select 128-bit RNS limbs, we will have fewer limbs to work with, resulting in a decrease in the number of available levels but result in high precision for an application. With our chosen parameters, using 128-bit wide RNS limbs would leave us with an insufficient number of limbs to perform a single bootstrapping operation. To strike a balance between performance and precision, we choose to use 64-bit wide RNS limbs in this work.

Most GPUs in the market natively support 16-, 32-, and 64-bit floating point computations as well as 4-, 8-, 32-bit integer computations. Unfortunately, they lack dedicated hardware support for 64-bit integer operations, the most common operation in FHE workloads. Instructions for processing 64-bit integer operands are emulated using multiple 32-bit integer instructions, making them comparatively slower. To complement our native modular reduction, which relies on 64-bit integer operations, we add support for hardware-backed 64-bit integer multiplier and accumulator, as well as widen the register-file size to accommodate the large ciphertexts. Table 4 demonstrates the decrease in total cycles for each of our proposed native modular instructions in comparison to the MI100 GPU-emulated instructions in the baseline (vanilla) configuration.

Prior studies [28, 84] argued that dedicating resources to specialized 64-bit integer cores was not justifiable in terms of opportunity cost, as workloads at the time did not necessitate INT64 support, and emulation with 32-bit cores was sufficient. However, in the context of FHE, we maintain that the performance improvements attained through using an upgraded vector ALU justify the additional chip resources allocated.

3.3 LABS: Locality-Aware Block Scheduler

So far, our microarchitectural extensions primarily focused on optimizing individual FHE blocks. To better leverage these new features, we focus next on inter-block optimization opportunities, targeting the workgroup dispatcher within the CDNA architecture. GPU scheduling is typically managed using streams of blocks that are scheduled on compute units in a greedy manner [9]. The presence of large GPU register files allows the scheduler to oversubscribe blocks to each compute unit. However, the existing scheduler within the CDNA architecture is not cognizant of inter-block data dependencies, forcing cache flushes when transitioning from one block to the next.

We propose a Locality-Aware Block Scheduler (LABS) designed to schedule blocks with shared data together, thus avoiding redundant on-chip cache flushes, specifically in the LDS. LABS further benefits from our set of microarchitectural enhancements, which relax the operational constraints during block scheduling and create new opportunities for optimization (for instance, the (**cNoC**) feature enables LDS data to be globally accessible across all CUs, thereby allowing the scheduler to assign blocks to any available CU). To develop LABS, we employ a well-known graph-based mapping solution and frame the problem of block mapping to CUs as a compile-time Graph Partitioning Problem (GPP) [80, 85].

Graph Partitioning Problem: To develop our locality-aware block scheduler, we use two graphs. Let $G = G(V, E)$ represent a directed acyclic compute graph with vertices V (corresponding to FHE blocks) and edges E (indicating the data dependencies of the blocks). Similarly, let $G_a = G_a(V_a, E_a)$ denote an undirected graph with vertices V_a (representing GPU compute units) and edges E_a

(illustrating the communication links between compute units). Both edge sets, E and E_a, are assumed to be weighted, with edge weights of E signifying the size of data transferred between related blocks, and E_a representing the bandwidth of communication between corresponding compute units. We can then define $\pi : V \rightarrow V_a$ as a mapping of V into V_a disjoint subsets. Our objective is to find a mapping π that minimizes communication overhead between compute units.

We formulate our Graph Partitioning Problem (GPP) by introducing a cost function Φ. For a graph G, if it is partitioned such that E_c denotes the set of edge cuts, then Φ can be expressed as the sum of the individual cut-edge weights (with (v, w) representing the edge-weight of the edge connecting node v to node w). The cost function Φ reflects the communication overhead associated with assigning FHE blocks to separate compute units. The goal of the graph partitioning problem is to discover a partition that evenly distributes the load across each compute unit while minimizing the communication cost Φ.

$$\Phi = |E_c| = \sum_{(v,w) \in E_c} |(v, w)|$$

In this equation, $|(v, w)|$ signifies the data transferred between FHE blocks. To partition the compute graph and prepare it for mapping onto the architecture graph, we utilize a multilevel mesh partitioning technique. For readers interested in gaining further insights into our graph partitioning implementation of the multilevel mesh partitioning algorithm, we recommend referring to the work of Walshaw and Cross [85].

Architecture-aware mapping: In this work, we focus on mapping our partitioned subgraphs onto the set of compute units V_a, where communication costs (both latency and bandwidth) are not uniformly distributed across the network [75]. To uniformly distribute the communication overheads across the network, we introduce a network cost function Γ. Here, Γ is defined as the product of individual cut-weights and their corresponding edge-weights in the architecture graph when mapped using a mapping function π. Formally, Γ is described as:

$$\Gamma = \sum_{(v,w) \in E_c} |(v, w)|.|(\pi(v), \pi(w))|$$

In this equation, $\pi(v)$ represents the mapping of block v to a compute unit from the set V_a, after applying the mapping function π. Additionally, $|(\pi(v), \pi(w))|$ represents the communication bandwidth between compute units $\pi(v)$ and $\pi(w)$. Similar to our analysis with Φ, our goal is to minimize Γ. To accomplish this, we use a compile-time optimization by applying *simulated annealing*, alongside mesh partitioning, to map FHE blocks onto compute units efficiently. The evaluation of performance improvements by incorporating the **LABS** is discussed further in Section 4.

4 EVALUATION

In this section, we first give a concise overview of the GPU simulator employed to model our microarchitectural extensions. Next, we outline the evaluation methodology assumed to assess the performance of our bootstrapping and other workload implementations. Finally, we present evaluation results.

Table 5: MI100 GPU Parameters

Parameter	Value
GPU Core Freq	1502 MHz
Process Size	7 nm
TFLOPS	23.07
Register File	15 MB
CU count	120
L1 Vector Cache	16 KB per CU
L1 Scalar Cache	16 KB
L1 Inst Cache	32 KB
Shared L2	8 MB
LDS	7.5 MB
GPU Memory	32 GB HBM2
Mem Bandwidth	1229 GB/s
Host CPU	AMD EPYC 7002
Host OS	Ubuntu 18.04
GPU Driver	AMD ROCm 5.2.5

4.1 The NaviSim and BlockSim Simulators

In our work, we leverage NaviSim [11], a cycle-level execution-driven GPU architecture simulator. NaviSim faithfully models the CDNA architecture by implementing a CDNA ISA emulator and a detailed timing simulator of all the computational components and memory hierarchy. NaviSim utilizes the Akita simulation engine [81] to enable modularity and high-performance parallel simulation. NaviSim is highly configurable and accurate and has been extensively validated against an AMD MI100 GPU. As an execution-driven simulator, NaviSim recreates the execution results of GPU instructions during simulation with the help of an instruction emulator for CDNA ISA [7, 12]. Currently, NaviSim supports kernels written in both OpenCL [43] and the HIP programming language [9]. For our experiments, we implement our kernels using OpenCL. NaviSim can generate a wide range of output data to facilitate performance analysis. For performance metrics related to individual components, NaviSim reports instruction counts, average latency spent accessing each level of cache, transaction counts for each cache, TLB transaction counts, DRAM transaction counts, and read/write data sizes. For low-level details, NaviSim can generate instruction traces and memory traces. Finally, NaviSim can produce traces using the Daisen format so that users can use Daisen, a web-based visualization tool [82], to inspect the detailed behavior of each component.

We enhance NaviSim's capabilities by incorporating our new custom kernel-level simulator, BlockSim. BlockSim is designed to enable us to identify inter-kernel optimization opportunities. With an adjustable sampling rate for performance metrics, BlockSim accelerates simulations, facilitating more efficient design space exploration. BlockSim generates analytical models of the FHE Blocks to provide estimates for run times of various GPU configurations. When the best design parameters are identified, NaviSim is then employed to generate cycle-accurate performance metrics. Besides supporting FHE workloads, BlockSim serves as an essential component of NaviSim by abstracting low-level implementation details from the user, allowing them to focus on entire workloads

Table 6: Architecture comparison of various FHE accelerators

Parameters	Lattigo	F1	BTS	CL	ARK	FAB	100x	T-FHE	**GME**	GME-cNoC	GME-MOD	GME-WMAC
Technology (nm)	14	12/14	7	12/14	7	16	12	7	**7**			
Word size (bit)	54	32	64	28	64	54	54	32	**54**			
On-chip memory (MB)	6	64	512	256	512	43	6	20.25	**15.5**			
Frequency (GHz)	3.5	1.0	1.2	1.0	1.0	0.3	1.2	1.4	**1.5**	1.68‡	1.63‡	1.72‡
Area (mm^2)	122	151.4	373.6	472.3	418.3	-	815	826	**700*** **+ 186.2†**	96.82	48.27	41.11
Power (W)	91	180.4	163.2	317	281.3	225	250	400	**300*** **+ 107.5†**	53.91	31.86	21.73

*The CDNA architecture-based MI100 GPU chip area and power consumption are not disclosed. We display the publicly available approximated values.
†We compute the chip area and power requirements of our microarchitectural extensions using RTL components and Cadence Synthesis Solutions with the ASAP7 technology library.
‡Reported values are of maximum clock frequency F_{max} that the design can sustain without violating timing constraints.

rather than individual kernels. BlockSim enables restructuring of the wavefront scheduler and integrates compile-time optimizations obtained from **LABS**. We utilize AMD's CDNA architecture-based MI100 GPU to create a baseline for FHE application evaluations. We further validate our BlockSim findings with the MI100 GPU.

4.2 Experimental Setup

In our experiments, we determine our baseline performance using an AMD MI100 CDNA GPU (see table 5). We then iteratively introduce microarchitectural extensions and evaluate the performance benefits of each enhancement. We first evaluate our three microarchitectural extensions (**cNoC**, **MOD**, **WMAC**), then evaluate our compile-time optimization **LABS**, and conclude with a memory size exploration to determine the impact of on-chip memory size on FHE workloads. We evaluate these microarchitectural enhancements and compiler optimization using NaviSim and BlockSim. To determine the power and area overhead of our proposed microarchitectural components, we implement them in RTL. Utilizing Cadence Genus Synthesis Solutions, we synthesize these RTL components targeting an ASAP7 technology library [22] and determine the area and power consumption for each proposed microarchitectural element.

We first evaluate our bootstrapping implementation performance, utilizing the *amortized mult time per slot* metric [41]. This metric has been used frequently in the past to perform a comparison between different bootstrapping implementations. We can compute this metric as follows:

$$\mathbf{T}_{A.S.} = \frac{\mathbf{T}_{\text{boot}} + \sum_{\ell=1}^{L-L_{\text{boot}}} \mathbf{T}_{\text{mult}}(\ell)}{L - L_{\text{boot}}} \cdot \frac{1}{n} \quad (1)$$

Here, $\mathbf{T}_{\text{boot}}$ stands for total bootstrapping runtime, and L_{boot} stands for the number of levels that the bootstrapping operation utilizes. The rest of the parameters are defined in Table 1. The parameters that we have used in our implementation have an $\mathrm{L}_{\text{boot}} = 17$ and $n = 2^{15}$. In addition, we analyze the performance of two workloads: HE-based logistic regression (HELR) [35] and encrypted ResNet-20 [50] utilizing the CIFAR-10 dataset. For all three workloads, we evaluate the contributions of each individual FHE building block (see Table 2) that make up the respective workload. In addition, for these workloads, we report the performance benefits achieved by employing each of the proposed microarchitectural enhancements.

Table 7: Performance of various FHE building blocks

	CMult	HE-Add	HE-Mult	Rotate	Rescale
HyPHEN-CPU [62] (μs)	506	202	17300	15500	3900
100x [41] (μs)	130	160	2960	2550	490
T-FHE [27] (μs)	46	37	1131	1008	77
Baseline MI100 (μs)	178	217	4012	3473	681
GME* (μs)	**22**	**28**	**464**	**364**	**69**
Speedup over HyPHEN	23×	7.2×	37.3×	42.6×	56.5×
Speedup over 100x	5.9×	5.7×	6.4×	7×	7.1×
Speedup over T-FHE	2.1×	1.3×	2.4×	2.8×	1.1×
Speedup over Baseline	8.1×	7.8×	8.6×	9.5×	9.9×

*The values displayed here exclude contributions from the LABS optimization, as LABS is an *inter-block* optimization, and the metrics provided are intended for individual blocks.

We also compare our implementations with other state-of-the-art CKKS accelerators, incorporating a diverse selection of CPU [16, 62], GPU [27, 41, 62], FPGA [1], and ASIC [44, 45, 69, 70] platforms.[1] Table 6 presents a detailed comparison of the key architectural parameters across all the related works. Table 6 also showcases the distribution of chip area and power requirements for each microarchitectural enhancement of GME. Since the maximum operating frequency F_{max} of our microarchitectural enhancements (1.63 GHz) is greater than the typical operating frequency of the MI100 GPU (1.5 GHz), we do not expect our extensions to change the critical path timings of the MI100 design. It is essential to emphasize that operating frequencies differ across various designs, a crucial factor to consider when comparing execution times in absolute terms. Moreover, the ASIC designs make use of large on-chip memory, resulting in an expensive solution, and they are also not as flexible as CPU, GPU, and FPGA.

[1]In this section, we refer to the CPU implementation as Lattigo, the GPU implementation as 100x, and the CraterLake ASIC design as CL. For the other accelerators, we use the full names from the respective papers.

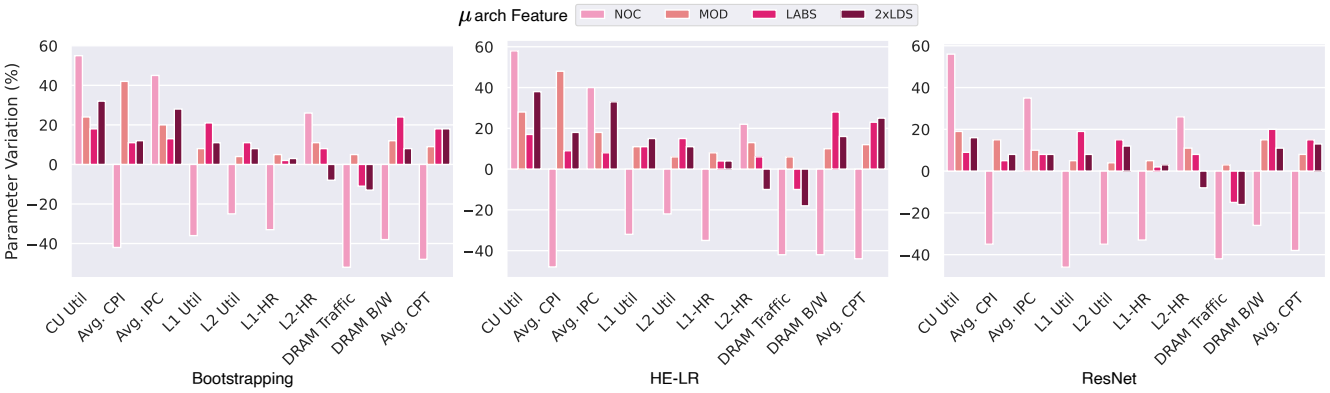


Figure 6: Influence of individual proposed microarchitectural extension on architectural performance metrics. Metrics illustrate a cumulative profile where each enhancement builds upon the preceding set of improvements

4.3 Results

Performance of FHE Building Blocks: We begin by comparing the performance of individual FHE blocks with the previous state-of-the-art GPU implementation [41]. Since these are individual FHE blocks, the reported metrics do not account for our inter-block **LABS** compiler optimization. We find that `HEMult` and `HERotate` are the most expensive operations, as they require key switching operations that involve the most data transfers from the main memory. The next most expensive operation is `HERescale`, where the runtime is dominated by the compute-intensive `NTT` operations.

Across the five FHE blocks mentioned in Table 7, we achieve an average speedup of 6.4× compared to the `100x` implementation. In particular, we see a substantial performance improvement in the most expensive operations, namely `HEMult` and `HERotate`, as our proposed microarchitectural enhancements reduce the data transfer time by 12× for both blocks. For `HERescale`, we manage to decrease the average memory transaction latency by 13× using our microarchitectural enhancements to the on-chip network, **cNoC**. Thus making `HERescale` the fastest block in comparison to `100x` GPU implementation.

Impact of Microarchitectural Extensions: Figures 6 and 7 highlight the impact of each of our proposed microarchitectural extensions as well as our compile-time optimizations across three different workloads, i.e., bootstrapping, HE-LR, and ResNet-20.

First, our proposed concentrated 2D torus network enables ciphertexts to be preserved in on-chip memory across kernels, leading to a significant increase in compute unit utilization across workloads, thereby reducing the average cycles consumed per memory transaction (see Avg. CPT in Figure 6). In fact, when comparing the average number of cycles spent per memory transaction (average CPT), we observe that the ResNet-20 workload consistently displays a lower average CPT value compared to the HE-LR workload. This indicates a higher degree of data reuse within the ResNet-20 workload across FHE blocks as opposed to the HE-LR workload. With **cNoC** enhancement, as the data required from previous kernels is retained in the on-chip memory, CUs are no longer starved for data and this also results in a substantial decrease in DRAM bandwidth utilization and DRAM traffic (the total amount of data transferred

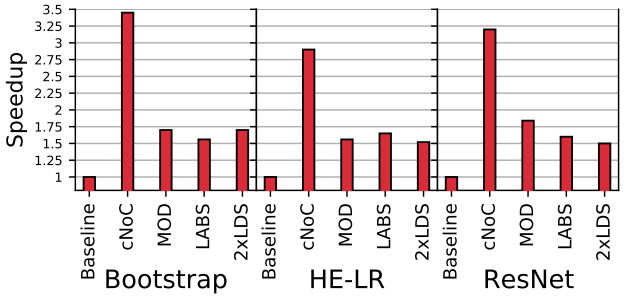


Figure 7: Speedup achieved from each microarchitectural extension. The baseline refers to a vanilla MI100 GPU. The reported speedup is cumulative, with each microarchitectural enhancement building upon the previous ones

from DRAM). The L1 cache utilization decreases notably across all three workloads for the **cNoC** microarchitectural enhancement. This is due to the fact that the LDS bypasses the L1 cache, and memory accesses to the LDS are not included in the performance metrics of the L1 cache.

The proposed **MOD** extension enhances the CDNA ISA by adding new instructions. These new instructions are complex instructions that implement commonly used operations in FHE, like `mod-red`, `mod-add`, and `mod-mult`. As these instructions are complex (composed of multiple sub-instructions), they consume a higher number of cycles than comparatively simpler instructions such as `mult` or `add`. This is the reason for the increase in the average cycles per instruction (CPI) metric shown in Figure 6.

The compile-time **LABS** optimization in our approach further removes redundant memory transactions by scheduling blocks that share data together, thus reducing total DRAM traffic and enhancing CU utilization. **LABS** takes advantage of the on-chip ciphertext preservation enabled by our **cNoC** microarchitectural enhancement. Across bootstrapping, HE-LR, and ResNet-20 workloads, **LABS** consistently delivers an additional speedup of over 1.5× on top of **cNoC** and **MOD** (See Figure 7).

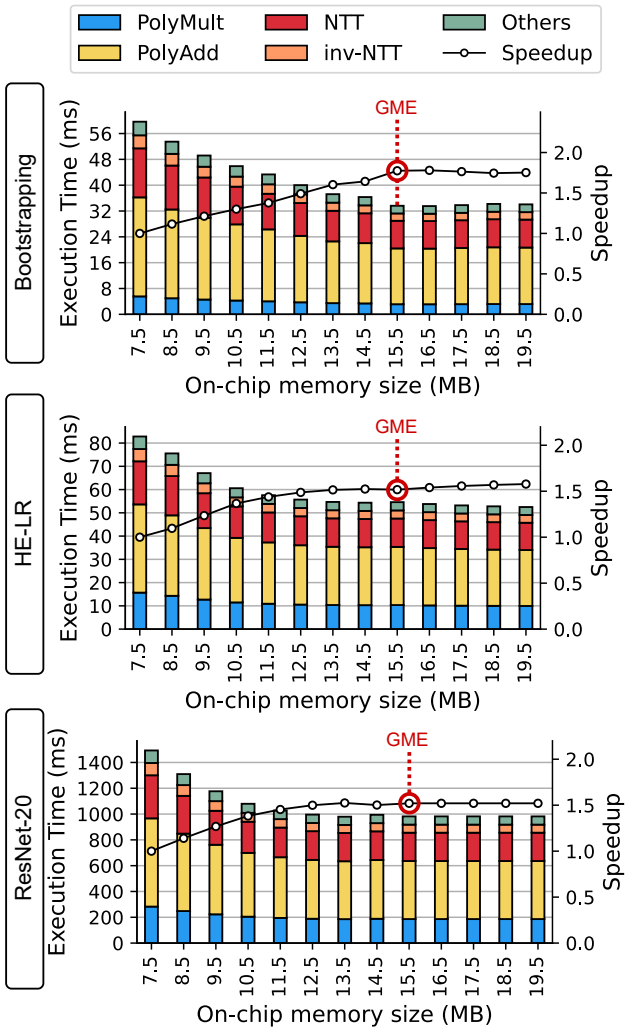


Figure 8: Exploring the impact of on-chip memory size on FHE workload performance

Performance Comparison: We compare the performance of GME with 100x implementation of FHE workloads in Table 8. GME surpasses the previous best GPU-based implementation for bootstrapping and HE-LR by factors of 15.7× and 14.2×, respectively. Note that we do not compare the performance of ResNet-20 workload with 100x, as they do not implement this workload. With close to double the on-chip memory (LDS), and similar peak memory bandwidth, our microarchitectural extensions paired with our compiler optimization delivered significant performance improvement across all three FHE workloads. GME significantly outperforms the CPU implementation Lattigo by 514×, 1165×, and 427× for bootstrapping, HE-LR, and ResNet-20 workloads, respectively. We assessed Lattigo's performance by executing workloads on an Intel 8th-generation Xeon Platinum CPU with 128 GB of DDR4 memory.

Table 8: HE workloads execution time comparison of proposed GME extensions with other architectures

Accelerator	Arch.	$T_{A.S.}$ (*ns*)	Boot (*ms*)	HE-LR (*ms*)	ResNet 20 (*ms*)
Lattigo [59]	CPU	8.8*e*4	3.9*e*4	23293	-
HyPHEN [62]	CPU	2110	2.1*e*4	-	3.7*e*4
F1 [69]	ASIC	2.6*e*5	Yes†	1024	-
BTS [45]	ASIC	45	58.9	28.4	1910
CL [70]	ASIC	17	4.5	15.2	321
ARK [44]	ASIC	14	3.7	7.42	125
FAB [1]	FPGA	470	92.4	103	-
100x [41]	V100	740	528	775	-
HyPHEN [62]	V100	-	830	-	1400
T-FHE [27]	A100	404	157	178	3793
Baseline	MI100	863	413	658	9989
GME	**MI100+**	**74.5**	**33.63**	**54.5**	**982**

† F1 is limited to a single-slot bootstrapping, while others support packed bootstrapping.

In addition, GME outperforms the FPGA design implementation of FHE workloads, called FAB [1], by 2.7× and 1.9× for bootstrapping and HE-LR workloads, respectively. A primary factor contributing to this acceleration is the low operating frequency of FPGAs (the Alveo U280 used in FAB operates at 300MHz, while GME cores can achieve peak frequencies of 1.5GHz [21]). In their work, FAB scales their implementation to 8 FPGAs for the HE-LR workload (referred to as FAB-2). GME surpasses FAB-2 by 1.4×. This occurs because, when the intended application cannot be accommodated on a single FPGA device, considerable communication overheads negate the advantages of scaling out.

However, GME does not outperform all ASIC implementations shown in Table 8. While it achieves an average speedup of 18.7× over F1 for the HE-LR workload, it falls short in comparison to BTS, CL, and ARK due to their large on-chip memory and higher HBM bandwidths. ASIC implementations are tailored for a single workload. Their customized designs lack flexibility, so they cannot easily accommodate multiple workloads across domains. Cutting-edge implementations such as ARK [44] integrate the latest HBM3 technology, enabling them to utilize nearly twice the memory bandwidth available in HBM3, as compared to HBM2 used on MI100 GPUs. CraterLake (CL) [70] incorporates extra physical layers (PHY) to facilitate communication between DRAM and on-chip memory, thereby enhancing the available bandwidth for FHE workloads. In this paper, we limit our focus to an existing HBM model compatible with the CDNA architecture without modifications to the physical communication layers.

On-chip Memory Size Exploration: Finally, we look for the ideal on-chip memory (LDS) size for the FHE workload, as shown in Figure 8. By increasing the total LDS size from 7.5MB (which is the current LDS size on MI100 GPU) to 15.5MB, we achieve speedups of 1.74×, 1.53×, and 1.51× for Bootstrapping, HE-LR, and ResNet-20 workloads, respectively. However, increasing the LDS size beyond 15.5 MB does not result in substantial speedup, as DRAM bandwidth becomes a bottleneck.

5 DISCUSSION

In the field of accelerator design, developing general-purpose hardware is of vital importance. Rather than creating a custom accelerator specifically for FHE, we focus on extending the capabilities of existing GPUs to take advantage of the established ecosystems for GPUs. General-purpose hardware, such as GPUs, reap the benefits of versatile use of all microarchitectural elements present on the GPU. In this section, we demonstrate the potential advantages of the proposed microarchitectural enhancements across various domains, confirming the importance of these microarchitectural features. Our observations are based on prior works, which highlight the potential benefits of similar optimizations across diverse workloads. We evaluate the influence of each optimization by examining communication overheads, high data reuse, utilizing modular reduction, or employing integer arithmetic. Table 9 presents an overview of our findings, highlighting the potential advantages of the proposed microarchitectural extensions across an array of other workloads.

The recent Hopper architecture by NVIDIA for the H100 GPU introduced a feature termed DSMEM (Distributed Shared Memory). This allows the virtual address space of shared memory to be logically spread out across various SMs (streaming multiprocessors) [26]. Such a configuration promotes data sharing between SMs, similar to the (**cNoC**) feature we introduced. However, the details of the SM-to-SM network for DSMEM are not publicly available and to the best of our knowledge, the SM-to-SM connectivity is not global but limited to the Thread Block Cluster comprised of 8 SMs. In contrast, the (**cNoC**) proposed by us enables global connectivity to all 120 CUs in our MI100 GPU, enabling efficient all-to-all communication. For enhancing FHE performance, it's crucial to substantially reduce the latency in SM-to-SM communication. We aim to conduct a detailed analysis comparing the inter-SM communication overheads of the H100 GPU to those of GME in future work.

Table 9: Potential benefits of proposed microarchitectural extensions across various workloads

Applications	NOC	MOD	WMAC	LABS
AES [36, 49]	✔	✔	✔	✔
FFT [25]	✔	✔	✔	✔
3D Laplace [74, 86]	✔	✘	✔	✔
BFS [18, 56]	✔	✘	✔	◆
K-Means [23]	✔	✘	✘	✔
ConvNet2 [53]	✔	✘	✔	◆
Transformer [37, 72]	✔	✘	✔	◆
Monte Carlo [52]	✘	✘	✔	✘
N-Queens [40]	✘	✘	✔	✔
Black-Scholes [32]	✘	✘	✔	✘
Fast Walsh [14]	✔	✘	✔	✔

✔ Proposed optimization has the potential to significantly improve workload performance.
✘ Proposed optimization is unlikely to result in notable performance improvements.
◆ Further experimentation is necessary, as it is uncertain whether the proposed optimization will lead to performance improvement

6 RELATED WORK

CPU/GPU implementations: Several algorithmic implementations, such as `Lattigo` [58], SEAL [73], HEXL [15], HEAAN [20], HELib [13, 34], and PALISADE [64], have recently been proposed for FHE using the CKKS scheme. Despite the efforts put forth by these libraries, a CPU-based implementation of FHE remains infeasible due to the relatively limited computational power of CPUs.

PRIFT [3] and the work by Badawi et al. [5] aims to accelerate FHE using NVIDIA GPUs. Although they support most HE blocks, they do not accelerate bootstrapping. `100x` [41] speeds up all HE blocks, including bootstrapping. While `100x` optimizes off-chip memory transactions through *kernel-fusions*, their implementation still results in redundant memory transactions due to partitioned on-chip memory of V100. Locality-aware block scheduling [51] has been proposed in GPUs to maximize locality within each core; however, LABS maximizes locality by exploiting the globally shared LDS through the proposed (**cNoC**).

FPGA accelerators: Multiple prior efforts [46, 47, 66, 68] have developed designs for FHE workloads. However, most of them either do not cover all HE primitives or only support smaller parameter sets that allow computation up to a multiplicative depth of 10. HEAX [66] is an FPGA-based accelerator that only speeds up CKKS encrypted multiplication, with the remainder offloaded to the host processor.

FAB demonstrates performance comparable to the previous GPU implementation, `100x` [41], and ASIC designs `BTS` [45] and `F1` [69] for certain FHE workloads. Although FPGAs show great potential for accelerating FHE workloads, they are limited by low operating frequencies and compute resources. Furthermore, the substantial communication overhead and the time required to program the FPGA discourages their wide-scale deployment [63].

ASIC accelerators: There exist several recent ASIC designs including F1 [69], CraterLake [70], BTS [45], and ARK [44] that accelerate the CKKS FHE scheme. F1 implementation makes use of small N and Q values, implementing only a single-slot bootstrapping. BTS is the first ASIC proposal demonstrating the performance of a fully-packed CKKS bootstrapping. CraterLake and ARK design further enhance the packed CKKS bootstrapping performance and demonstrate several orders of performance improvement across various workloads.

7 CONCLUSION

In this work, we present an ambitious plan for extending existing GPUs to support FHE. We propose three novel microarchitectural extensions followed by compiler optimization. We suggest a 2D torus on-chip network that caters to the all-to-all communication patterns of FHE workloads. Our native modular reduction ISA extension reduces the latency of modulus reduction operation by 43%. We enable native support for 64-bit integer arithmetic to mitigate math pipeline throttling. Our proposed BlockSim simulator enhances the capabilities of the open-source GPU simulator, NaviSim, allowing for coarse-grained simulation for faster design space exploration. Overall, comparing against previous state-of-the-art GPU implementations [41], we obtain an average speedup of 14.6× across workloads as well as outperform the CPU, the FPGA, and some ASIC implementations.

ACKNOWLEDGMENTS

This research was supported in part by the Institute for Experiential AI and the NSF IUCRC Center for Hardware and Embedded Systems Security and Trust (CHEST), NSF CNS 2312275, NSF CNS 2312276, and by Samsung Advanced Institute of Technology, Samsung Electronics Co., Ltd. Additionally, we acknowledge the financial assistance from grant RYC2021-031966-I funded by MCIN/AEI/10.13039/501100011033 and the "European Union NextGenerationEU/PRTR."

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895. https://doi.org/10.1109/HPCA56546.2023.10070953

[2] Rashmi Agrawal and Ajay Joshi. 2023. On Architecting Fully Homomorphic Encryption-based Computing Systems. https://doi.org/10.1007/978-3-031-31754-5

[3] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. 2020. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access* 8 (2020), 226544–226556.

[4] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. 2020. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 379–391.

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. 2018. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 70–95.

[6] AMD 2020. *AMD Instinct MI100 Instruction Set Architecture*. AMD. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi100-cdna1-shader-instruction-set-architecture.pdf Reference Guide.

[7] AMD Inc. 2020. "AMD Instinct MI100" Instruction Set Architecture, Reference Guide. https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf

[8] AMD Inc. 2020. Introducing CDNA Architecture, The All-New AMD GPU Architecture for the Modern Era of HPC & AI. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

[9] AMD Inc. 2022. HIP Programming Guide. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html

[10] James Balfour and William J Dally. 2006. Design tradeoffs for tiled CMP on-chip networks. In *ACM International conference on supercomputing 25th anniversary volume*. 390–401. https://doi.org/10.1145/2591635.2667187

[11] Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2022. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 333–345. https://doi.org/10.1145/3559009.3569666

[12] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. 2021. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 13–23. https://doi.org/10.1109/ISPASS51385.2021.00013

[13] Flavio Bergamaschi. [n. d.]. HELib. https://github.com/homenc/HElib

[14] Dusan Bikov and Iliya Bouyukliev. 2018. Parallel fast Walsh transform algorithm and its implementation with CUDA on GPUs. *Cybernetics and Information Technologies* 18, 5 (2018), 21–43. https://eprints.ugd.edu.mk/id/eprint/20026

[15] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. 2021. Intel HEXL: accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 57–62. https://doi.org/10.1145/3474366.3486926

[16] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*. Springer, 587–617.

[17] C Bunn, Harrison Barclay, A Lazarev, F Yusuf, J Fitch, J Booth, Kaustubh Shivdikar, and D Kaeli. 2019. Student cluster competition 2018, team northeastern university: Reproducing performance of a multi-physics simulations of the Tsunamigenic 2004 Sumatra Megathrust earthquake on the AMD EPYC 7551 architecture. *Parallel Comput.* 90 (2019), 102568. https://doi.org/10.1016/j.parco.2019.102568

[18] Federico Busato and Nicola Bombieri. 2014. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2014), 1826–1838. https://doi.org/10.1109/TPDS.2014.2330597

[19] Jung Hee Cheon, Kyoohyung Han, and Duhyeong Kim. 2020. Faster Bootstrapping of FHE over the Integers. In *Information Security and Cryptology–ICISC 2019: 22nd International Conference, Seoul, South Korea, December 4–6, 2019, Revised Selected Papers*. Springer, 242–259. https://doi.org/10.1007/978-3-030-40921-0_15

[20] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. 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*. Springer, 409–437.

[21] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. 2020. When hls meets fpga hbm: Benchmarking and bandwidth optimization. *arXiv preprint arXiv:2010.06075* (2020). https://doi.org/10.48550/arXiv.2010.06075

[22] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. https://doi.org/10.1016/j.mejo.2016.04.006

[23] Salvatore Cuomo, Vincenzo De Angelis, Gennaro Farina, Livia Marcellino, and Gerardo Toraldo. 2019. A GPU-accelerated parallel K-means algorithm. *Computers & Electrical Engineering* 75 (2019), 262–274. https://doi.org/10.1016/j.compeleceng.2017.12.002

[24] Leo de Castro, Rashmi Agrawal, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. 2021. Does fully homomorphic encryption need compute acceleration? *arXiv preprint arXiv:2112.06396* (2021). https://doi.org/10.48550/arXiv.2112.06396

[25] Xinqiang Ding, Yujin Wu, Yanming Wang, Jonah Z Vilseck, and Charles L Brooks III. 2020. Accelerated CDOCKER with GPUs, parallel simulated annealing, and fast Fourier transforms. *Journal of chemical theory and computation* 16, 6 (2020), 3910–3919. https://doi.org/10.1021/acs.jctc.0c00145

[26] Anne C Elster and Tor A Haugdahl. 2022. Nvidia hopper gpu and grace cpu highlights. *Computing in Science & Engineering* 24, 2 (2022), 95–100. https://doi.org/10.1109/MCSE.2022.3163817

[27] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934. https://doi.org/10.1109/HPCA56546.2023.10071017

[28] Zhuo Feng, Zhiyu Zeng, and Peng Li. 2010. Parallel on-chip power distribution network analysis on multi-core-multi-GPU platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19, 10 (2010), 1823–1836. https://doi.org/10.1109/TVLSI.2010.2059718

[29] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. 2022. BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2205.14017* (2022). https://doi.org/10.48550/arXiv.2205.14017

[30] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[31] Craig Gentry and Shai Halevi. 2011. Implementing gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15–19, 2011. Proceedings 30*. Springer, 129–148.

[32] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating financial applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 127–136. https://doi.org/10.1145/2458523.2458536

[33] Saransh Gupta, Rosario Cammarota, and Tajana Šimunić Rosing. 2022. Memfhe: End-to-end computing with fully homomorphic encryption in memory. *ACM Transactions on Embedded Computing Systems* (2022). https://doi.org/10.1145/3569955

[34] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 554–571.

[35] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 9466–9471.

[36] Keisuke Iwai, Takakazu Kurokawa, and Naoki Nisikawa. 2010. AES encryption implementation on CUDA GPU and its analysis. In *2010 First International Conference on Networking and Computing*. IEEE, 209–214. https://doi.org/10.1109/ICNC.2010.49

[37] Mojan Javaheripi, Gustavo de Rosa, Subhabrata Mukherjee, Shital Shah, Tomasz Religa, Caio Cesar Teodoro Mendes, Sebastien Bubeck, Farinaz Koushanfar, and Debadeepta Dey. 2022. LiteTransformerSearch:

Training-free Neural Architecture Search for Efficient Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24254–24267. https://proceedings.neurips.cc/paper_files/paper/2022/hash/9949e6906be6448230cdba9a4cb2d564-Abstract-Conference.html

[38] Malith Jayaweera, Kaustubh Shivdikar, Yanzhi Wang, and David Kaeli. 2021. JAXED: Reverse Engineering DNN Architectures Leveraging JIT GEMM Libraries. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 189–202. https://doi.org/10.1109/SEED51797.2021.00030

[39] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. On-chip networks. *Synthesis Lectures on Computer Architecture* 12, 3 (2017), 1–210. https://picture.iczhiku.com/resource/eetop/SYieGarAzskjOvnm.pdf

[40] Cao Jianli, Chen Zhikui, Wang Yuxin, and Guo He. 2020. Parallel genetic algorithm for N-Queens problem based on message passing interface-compute unified device architecture. *Computational Intelligence* 36, 4 (2020), 1621–1637. https://doi.org/10.1111/coin.12300

[41] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[42] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access* 9 (2021), 98772–98789. https://doi.org/10.1109/ACCESS.2021.3096189

[43] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. 2015. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, Burlington,MA,USA. https://dahlan.unimal.ac.id/files/ebooks2/2015%203rd%20Heterogeneous%20Computing%20with%20OpenCL%202.0.pdf

[44] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1237–1254.

[45] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 711–725. https://doi.org/10.1145/3470496.3527415

[46] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Jung Hee Cheon, and Rob A Rutenbar. 2019. FPGA-based accelerators of fully pipelined modular multipliers for homomorphic encryption. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[47] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Yujin Nam, Jung Hee Cheon, and Rob A Rutenbar. 2020. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 56–64.

[48] Miroslav Knezevic, Frederik Vercauteren, and Ingrid Verbauwhede. 2010. Faster interleaved modular multiplication based on Barrett and Montgomery reduction methods. *IEEE Trans. Comput.* 59, 12 (2010), 1715–1721. https://doi.org/10.1109/TC.2010.93

[49] Deguang Le, Jinyi Chang, Xingdou Gou, Ankang Zhang, and Conglan Lu. 2010. Parallel AES algorithm for fast data encryption on GPU. In *2010 2nd international conference on computer engineering and technology*, Vol. 6. IEEE, V6–1. https://doi.org/10.1109/ICCET.2010.5486259

[50] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning*. PMLR, 12403–12422.

[51] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 260–271. https://doi.org/10.1109/HPCA.2014.6835937

[52] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*. 451–460. https://doi.org/10.1145/1815961.1816021

[53] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance analysis of GPU-based convolutional neural networks. In *2016 45th International conference on parallel processing (ICPP)*. IEEE, 67–76. https://doi.org/10.1109/ICPP.2016.15

[54] Neal Livesay, Gilbert Jonatan, Evelio Mora, Kaustubh Shivdikar, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2023. Accelerating finite field arithmetic for homomorphic encryption on GPUs. *2023 IEEE MICRO* (2023). https://doi.org/10.1109/MM.2023.3253052

[55] Souhail Meftah, Benjamin Hong Meng Tan, Khin Mi Mi Aung, Lu Yuxiao, Lin Jie, and Bharadwaj Veeravalli. 2022. Towards high performance homomorphic encryption for inference tasks on CPU: An MPI approach. *Future Generation Computer Systems* 134 (2022), 13–21.

[56] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[57] Daniele Micciancio and Oded Regev. 2009. Lattice-based cryptography. *Post-quantum cryptography* (2009), 147–191.

[58] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. 64–70.

[59] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2022. Lattigo v4. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

[60] OpenAI. 2023. March 20 CHATGPT outage: Here's what happened. https://openai.com/blog/march-20-chatgpt-outage

[61] Ali Şah Özcan, Can Ayduman, Enes Recep Türkoğlu, and Erkay Savaş. 2023. Homomorphic Encryption on GPU. *IEEE Access* (2023).

[62] Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. HyPHEN: A Hybrid Packing Method and Optimizations for Homomorphic Encryption-Based Neural Network. (2023). https://doi.org/10.48550/arXiv.2302.02407

[63] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743. https://doi.org/10.1109/ACCESS.2020.3012084

[64] Yuriy Polyakov. [n. d.]. Palisade Library. https://gitlab.com/palisade/palisade-release

[65] Thomas Pöppelmann, Tobias Oder, and Tim Güneysu. 2015. High-Performance Ideal Lattice-Based Cryptography on 8-Bit ATxmega Microcontrollers. In *Progress in Cryptology—LATINCRYPT*. Springer, 346–365. https://doi.org/10.1145/3092951

[66] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. 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*. 1295–1309.

[67] Sujoy Sinha Roy, Ahmet Can Mert, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, et al. 2021. Accelerator for computing on encrypted data. *Cryptology ePrint Archive* (2021).

[68] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 387–398.

[69] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252. https://doi.org/10.1145/3466752.3480070

[70] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[71] Mohanad Sarhan, Siamak Layeghy, Marcus Gallagher, and Marius Portmann. 2023. From zero-shot machine learning to zero-day attack detection. *International Journal of Information Security* (2023), 1–13. https://link.springer.com/article/10.1007/s10207-023-00676-0

[72] Teven Le Scao, Thomas Wang, Daniel Hesslow, Lucile Saulnier, Stas Bekman, M Saiful Bari, Stella Bideman, Hady Elsahar, Niklas Muennighoff, Jason Phang, et al. 2022. What Language Model to Train if You Have One Million GPU Hours? *arXiv preprint arXiv:2210.15424* (2022). https://doi.org/10.48550/arXiv.2210.15424

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

[74] Jungkyun Shin, Wansoo Ha, Hyunggu Jun, Dong-Joo Min, and Changsoo Shin. 2014. 3D Laplace-domain full waveform inversion using a single GPU card. *Computers & Geosciences* 67 (2014), 1–13. https://doi.org/10.1016/j.cageo.2014.02.006

[75] Kaustubh Shivdikar. 2021. *SMASH: Sparse Matrix Atomic Scratchpad Hashing*. Ph. D. Dissertation. https://www.researchgate.net/publication/352018010_SMASH_Sparse_Matrix_Atomic_Scratchpad_Hashing Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-07.

[76] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. 2022. Accelerating polynomial multiplication for homomorphic encryption on GPUs. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 61–72. https://doi.org/10.1109/SEED55351.2022.00013

[77] Kaustubh Shivdikar, Ahan Kak, and Kshitij Marwah. 2015. Automatic image annotation using a hybrid engine. In *2015 Annual IEEE India Conference (INDICON)*. IEEE, 1–6. https://doi.org/10.1109/INDICON.2015.7443338

[78] Kaustubh Shivdikar, Kaushal Paneri, and David Kaeli. [n. d.]. Speeding up DNNs using HPL based Fine-grained Tiling for Distributed Multi-GPU Training.

([n. d.]).

[79] Victor Shoup. 2009. *A computational introduction to number theory and algebra*. Cambridge University Press. https://shoup.net/ntb/ntb-v2.pdf

[80] Mohit Srinivasan, Ahan Kak, Kaustubh Shivdikar, and Chirag Warty. 2016. Dynamic power allocation using Stackelberg game in a wireless sensor network. In *2016 IEEE Aerospace Conference*. IEEE, 1–10. https://doi.org/10.1109/AERO.2016.7500918

[81] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. 2019. MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 197–209. https://doi.org/10.1145/3307650.3322230

[82] Yifan Sun, Yixuan Zhang, Ali Mosallaei, Michael D Shah, Cody Dunne, and David Kaeli. 2021. Daisen: A Framework for Visualizing Detailed GPU Execution. *Eurographics Conference on Visualization* 40, 3 (2021), 239–250.

[83] Swadhin Thakkar, Kaustubh Shivdikar, and Chirag Warty. 2017. Video steganography using encrypted payload for satellite communication. In *2017 IEEE Aerospace Conference*. IEEE, 1–11. https://doi.org/10.1109/AERO.2017.7943978

[84] Ananta Tiwari, Kristopher Keipert, Adam Jundt, Joshua Peraza, Sarom S Leang, Michael Laurenzano, Mark S Gordon, and Laura Carrington. 2015. Performance and energy efficiency analysis of 64-bit ARM using GAMESS. In *Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing*. 1–10. https://doi.org/10.1145/2834899.2834905

[85] Chris Walshaw and Mark Cross. 2001. Multilevel mesh partitioning for heterogeneous communication networks. *Future generation computer systems* 17, 5 (2001), 601–623. https://doi.org/10.1016/S0167-739X(00)00107-2

[86] Lei Xiao, Guoxiang Yang, Kunyang Zhao, and Gang Mei. 2019. Efficient parallel algorithms for 3D Laplacian smoothing on the GPU. *Applied Sciences* 9, 24 (2019), 5437. https://doi.org/10.3390/app9245437

[87] Runhua Xu, Nathalie Baracaldo, and James Joshi. 2021. Privacy-preserving machine learning: Methods, challenges and directions. *arXiv preprint arXiv:2108.04417* (2021). https://doi.org/10.48550/arXiv.2108.04417

[88] Tian Ye, Rajgopal Kannan, and Viktor K Prasanna. 2022. FPGA Acceleration of Fully Homomorphic Encryption over the Torus. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.