University of Central Florida

STARS

Graduate Thesis and Dissertation 2023-2024

2024

Addressing Challenges in Utilizing GPUs for Accelerating Privacy-Preserving Computation

Ardhi Wiratama Baskara Yudha
University of Central Florida

Part of the Computer and Systems Architecture Commons
Find similar works at: https://stars.library.ucf.edu/etd2023
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Graduate Thesis and Dissertation 2023-2024 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Yudha, Ardhi Wiratama Baskara, "Addressing Challenges in Utilizing GPUs for Accelerating Privacy-Preserving Computation" (2024). *Graduate Thesis and Dissertation 2023-2024*. 103.
https://stars.library.ucf.edu/etd2023/103

ADDRESSING CHALLENGES IN UTILIZING GPUS FOR ACCELERATING PRIVACY-PRESERVING COMPUTATION

by

ARDHI WIRATAMA BASKARA YUDHA
M.S. University of Central Florida, 2023

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2024

Major Professor: Yan Solihin

© 2024 Ardhi Wiratama Baskara Yudha

ABSTRACT

Cloud computing increasingly handles confidential data, like private inference and query databases. Two strategies are used for secure computation: (1) employing CPU Trusted Execution Environments (TEEs) like AMD SEV, Intel SGX, or ARM TrustZone, and (2) utilizing emerging cryptographic methods like Fully Homomorphic Encryption (FHE) with libraries such as HElib, Microsoft SEAL, and PALISADE. To enhance computation, GPUs are often employed. However, using GPUs to accelerate secure computation introduces challenges addressed in three works.

In the first work, we tackle GPU acceleration for secure computation with CPU TEEs. While TEEs perform computations on confidential data, extending their capabilities to GPUs is essential for leveraging their power. Existing approaches assume co-designed CPU-GPU setups, but we contend that co-designing CPU and GPU is difficult to achieve and requires early coordination between CPU and GPU manufacturers. To address this, we propose software-based memory encryption for CPU-GPU TEE co-design via the software layer. Yet, this introduces issues due to AES's 128-bit granularity. We present optimizations to mitigate these problems, resulting in execution time overheads of 1.1% and 56% for regular and irregular applications.

In the second work, we focus on GPU acceleration for the CPU FHE library HElib, particularly for comparison operations on encrypted data. These operations are vital in Machine Learning, Image Processing, and Private Database Queries, yet their acceleration is often overlooked. We extend HElib to harness GPU acceleration for its resource-intensive components like BluesteinNTT, BluesteinFFT, and Element-wise Operations. Addressing memory separation, dynamic allocation, and parallelization challenges, we employ several optimizations to address these challenges. With all optimizations and hybrid CPU-GPU parallelism, we achieve a 11.1× average speedup over the state-of-the-art CPU FHE library.

In our latest work, we concentrate on minimizing the ciphertext size by leveraging insights from algorithms, data access patterns, and application requirements to reduce the operational footprint of an FHE application, particularly targeting Neural Network inference tasks. Through the implementation of all three levels of ciphertext compression (precision reduction in comparisons, optimization of access patterns, and adjustments in data layout), we achieve a remarkable 5.6$\times$ speedup compared to the state-of-the-art GPU implementation in 100x[30].

Overcoming these challenges is crucial for achieving significant GPU-driven performance improvements. This dissertation provides solutions to these hurdles, aiming to facilitate GPU-based acceleration of confidential data computation.

This achievement is dedicated to my mother.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Yan Solihin, for providing me with the opportunity to embark on this PhD journey and for his unwavering support throughout. His guidance has not only shaped my academic growth but has also enriched my ability to think critically and communicate effectively on technical subjects.

I would also like to extend my heartfelt thanks to the other members of my committee, Prof. Qian Lou, Prof. Huiyang Zhou, and Prof. Cliff Zou. Their guidance and feedback have played a crucial role in refining my dissertation work.

A special thanks goes out to all the current and past members of the ARPERS lab. It has been a collective journey, and I am grateful for the camaraderie and support we have shared along the way.

Last but certainly not least, I want to express my deepest appreciation to my wife for her unwavering support and understanding. Her assistance in managing household responsibilities and caring for our children, Hafshoh and Ibrahim, has been invaluable, especially during times when my focus was on my PhD work.

TABLE OF CONTENTS

LIST OF FIGURES xii

LIST OF TABLES xvi

CHAPTER 1: INTRODUCTION 1

Statement of Research 2

LITE: a low-cost practical inter-operable GPU TEE 2

BoostCom: Boosting the Comparison Operation on Encrypted Data using GPU . . 2

CHAPTER 2: LITERATURE REVIEW 4

GPU Trusted Execution Environment (TEE) 4

Fully Homomorphic Encryption (FHE) Acceleration 6

CHAPTER 3: LITE: A LOW-COST PRACTICAL INTER-OPERABLE GPU TEE 9

Introduction 9

Background 12

GPU Architecture and Unified Memory 12

Host-side TEE 13

Threat and Trust Model, Scope of Work 15
The Design of LITE 17
Rationale and Overview 17
Encryption APIs 19
AES Encryption and Address as the Tweak 23
Kernel Code Adaptation 24
Hardware Support 27
Code Optimizations 27
Masked shuffle 27
Delayed Shuffle 29
Selective Padding 30
Methodology 30
Evaluation 32
LITE Performance Overhead 32
Benefit of Partial Encryption on LITE 36
Performance of Software Implementation of AES 37
Performance Comparison between LITE and PSSM 38

Conclusion 39

CHAPTER 4: BOOSTCOM: BOOSTING THE COMPARISON OPERATION ON ENCRYPTED DATA USING GPU 40

Introduction 40

Background 45

BGV Scheme 45

Efficient Representation of Polynomial and NTT 46

Comparison Operation Algorithm 47

Bottleneck Analysis of Comparison Operation 48

The Design of BoostCom 51

Comparison-Friendly BGV Parameter Tuning 51

Hybrid CPU/GPU Parallelization 53

Slot Compaction 55

Non-Blocking Comparison 58

BluesteinNTT Acceleration 60

BluesteinFFT Acceleration 62

Accelerating Element-Wise Operations 62

Methodology 63
Experiment Platforms 63
Workload Evaluation Methodology 64
Microbenchmark 65
Applications 66
Evaluation Results 67
Workloads Speedup 67
Comparison Operation Speedup 68
BluesteinNTT and BluesteinFFT Sensitivity Study 70
Element-wise Sensitivity Study 72
Non-blocking Comparison Sensitivity Study 72
Conclusion 72
CHAPTER 5: BOOSTING FHE COMPUTATION VIA CIPHERTEXT COMPRESSION 73
Introduction 73
Background 74
Encryption Algorithm 74
Homomorphic Operation 75

Architecture Design 76

Overall Architecture 76

Data Flow of a Ciphertext 77

Reduced Precision of Comparison Operation 79

Load/Store Interface for Read-Only Ciphertext 79

Load/Store Interface for Data Layout Modification 80

Homomorphic Operation fusion and Lazy Operation 81

Methodology 83

Evaluation 84

Homomorphic Operation Performance Speedup 84

Reduced Precision of Homomorphic Comparison 84

Speedup on Neural Network Application 85

Conclusion 87

CHAPTER 6: THESIS CONCLUSION 88

APPENDIX A: COPYRIGHT INFORMATION 90

LIST OF REFERENCES 95

LIST OF FIGURES

Figure 3.1: Execution time overheads of memory security with a state-of-the-art scheme (PSSM [68]) and domain crossing re-encryption overheads. The former was collected from GPU simulation with machine configuration from [68], while the latter is from a real machine described in Section 3. 10

Figure 3.2: The block diagram of the AES-XTS encryption mode used in Intel MKTME. 14

Figure 3.3: The LITE vs Hardware-based GPU TEE typical data flow from host to device. 18

Figure 3.4: The timeline comparison between unoptimized shfl vs masked shfl. Following a divergent branch, only threads in path A fetch ciphertext from global memory. 28

Figure 3.5: The timeline comparison between unoptimized code and optimized code with delayed shuffle. Fetch A and fetch B are two independent global memory accesses. Unoptimized code decrypts the data immediately while delayed shuffle decrypts the data after the data are loaded into shared memory/registers. 29

Figure 3.6: Streaming Multiprocessor (SM) utilization and DRAM bandwidth utilization. 32

Figure 3.7: Kernel execution time overheads of naive LITE (shfl-naive+encryption) vs. optimized LITE (shfl-opt+encryption) over unsecured GPU baseline on regular GPU benchmarks. 33

Figure 3.8: Kernel execution time overhead of naive LITE implementation (pad+encryption) vs. optimized LITE (selective_pad+encryption) over unsecured GPU baseline on irregular GPU workloads. 35

Figure 3.9: Comparison of the L2 miss rate of unsecure GPU (baseline) vs. full padding (pad) vs. selective padding (selective_pad) on irregular GPU code. 36

Figure 3.10: Normalized execution time overhead over full encryption of optimized LITE for input-only encryption vs output-only encryption on irregular GPU code. 37

Figure 3.11: Comparison of execution time overhead of LITE vs. PSSM with re-encryption overhead evaluated in GPGPU-sim. 38

Figure 4.1: The workflow of performing computations on encrypted data that is transferred to an untrusted server using FHE. 41

Figure 4.2: Breakdown of BGV comparison time for Bivariate circuit with parameters $m = 34511$, $p = 3$, and $d = 6$. 49

Figure 4.3: Execution time breakdown for various BGV parameters for Bivariate (left) and Univariate (right) circuits. 50

Figure 4.4: Comparison of the homomorphic operation latencies of using the power of two vs. non-power of two vs. optimized non-power of two polynomial rings. 52

Figure 4.5: Illustrating Boostcom's hybrid parallelism: digits are computed across multiple CPU threads, while primitive operations in each digit are offloaded to the GPU. 55

Figure 4.6: Illustration of: (a) ResNet block containing convolution (Conv), batch normalization (BN), and ReLU; (b) Convolution filter; (c) Convolution steps on encrypted data resulting in unused slots; (d) Naive digit decomposition with many unused slots; and (e) Optimized digit decomposition with slot compaction. 57

Figure 4.7: Illustrating the saved cycles due to the non-blocking comparison optimization. 60

Figure 4.8: Layout optimization for offloading the element-wise operation to the GPU, utilizing additional copying at the CPU side to maximize the CPU-GPU *memcpy* bandwidth and parallelization degree. 64

Figure 4.9: The acceleration achieved by BoostCom in comparison to the baseline for five important workloads. 67

Figure 4.10: Speedups of the comparison operation for the Bivariate (top) and Univariate (bottom) circuit over the 16-core CPU baseline. 69

Figure 4.11: The comparison between BluesteinNTT and BluesteinFFT speedup over each baseline with the increasing parameter m. 70

Figure 4.12: Element-wise operation speedups of BoostCom's whole matrix approach vs. row-by-row GPU offloading, as Q (number of bits) increases. 71

Figure 4.13: Speedups of optimizations without vs. with non-blocking as the branch evaluation computation increases with larger exponent values. 71

Figure 5.1: Three-level ciphertext compression based on application knowledge, algorithm knowledge, and data layout modification. 76

Figure 5.2: Data flow from memory to ALU (top) of a compressed ciphertext. Data fetched from memory to L2 cache is an array of bits. The converter module (bits to int) converts the array into integers. PRNG generates the read-only ciphertext part based on a seed. 78

Figure 5.3: Steps to determine the minimum number of digits for acceptable accuracy. . 79

Figure 5.4: Timeline diagram depicting the matrix loads and stores for the sequential operations of PMULT, HADD, and PADD. Combining these operations will reduce the number of loads/stores for the matrix, as highlighted by the red oval lines. 81

Figure 5.5: Timeline diagram illustrating the online and offline stages of the PMULT, HADD, and PADD operations, with the offline parts delayed to conserve matrix loads. 82

Figure 5.6: Comparison of the speedup achieved between TensorFHE, Our Work, and Our Work+ relative to the baseline (100x) for CNN applications. 86

Figure 5.7: Speedup in execution time compared to the baseline for CNN application with varying degrees of precision reduction. 86

LIST OF TABLES

Table 2.1: Comparing LITE with prior GPU enclaves. 5

Table 2.2: The comparison of BoostCom vs. prior works. 7

Table 3.1: Encryption APIs 20

Table 3.2: Benchmarks 31

Table 4.1: Parameters used in BGV and comparison operation. 45

Table 4.2: Parameters and Statistics 65

Table 4.3: Memory Usage for each Workload (GB). 67

Table 5.1: Homomorphic Operation Building Block of BGV Scheme. 75

Table 5.2: Pseudo-Random Number Generator APIs 80

Table 5.3: Data Layout Converter APIs 80

Table 5.4: GPU Configuration. 82

Table 5.5: Practical FHE parameter used for homomorphic operation. 83

Table 5.6: Homomorphic Operation Speedup over the Baseline [30]. 84

Table 5.7: Number of Homomorphic Operations for Each Comparison Operation with Different Precision and Error Rates. 85

Table 5.8: The Accuracy of Neural Network Inference with Varying Comparison Precision. 87

CHAPTER 1: INTRODUCTION

Cloud computing is increasingly used to process sensitive data. Computing sensitive data requires careful consideration due to the potential security risks involved. Secure computation in the data center could be addressed with two different techniques: one involves utilizing a Trusted Execution Environment (TEE), such as Intel SGX, AMD SEV, or ARM TrustZone, while the other employs the novel cryptographic method of Fully Homomorphic Encryption (FHE). Each of these techniques offers distinct advantages and challenges in ensuring the confidentiality and integrity of the processed data.

Accelerators are commonly employed to enhance data center computations, with the GPU emerging as a pivotal component. Notably, recent strides in machine learning, exemplified by Deep Neural Networks (DNNs), have harnessed the substantial computational prowess of GPUs. Their capacity to manage formidable computational requirements extends to diverse scientific applications as well. The GPU is composed of numerous smaller, specialized cores. Collaboratively, these cores yield substantial performance gains, especially when processing tasks can be partitioned and executed across multiple cores in parallel.

To achieve accelerated computation of confidential data using GPUs, we face two options: extending the CPU Trusted Execution Environment (TEE) to incorporate GPU usage or optimizing the industry-standard Fully Homomorphic Encryption (FHE) library to efficiently offload its most resource-intensive components to the GPU. In line with these strategies, the ensuing chapters of this thesis present solutions tailored to tackle the arising challenges.

Statement of Research

LITE: a low-cost practical inter-operable GPU TEE

There is a strong need for GPU trusted execution environments (TEEs) as GPU is increasingly used in the cloud environment. However, current proposals either ignore memory security (i.e., not encrypting memory) or impose a separate memory encryption domain from the host TEE, causing a very substantial slowdown for communicating data from/to the host.

In this paper, we propose a flexible GPU memory encryption design called LITE that relies on software memory encryption aided by small architecture support. LITE's flexibility allows GPU TEE to be co-designed with CPU to create a unified encryption domain. We show that GPU applications can be adapted to the use of LITE encryption APIs without major changes. Through various optimizations, we show that software memory encryption in LITE can produce negligible performance overheads (1.1%) for regular benchmarks and still-acceptable overheads (56%) for irregular benchmarks.

BoostCom: Boosting the Comparison Operation on Encrypted Data using GPU

Fully Homomorphic Encryption (FHE) enables computation directly on encrypted data, ensuring data privacy. While prior efforts focused on accelerating linear operations, they ignored crucial non-linear integer operations like comparisons ($x < y$ and $x = y$), used in neural networks, image processing, and private queries, even though they are much slower than linear operations.

This paper introduces BoostCom, a scheme for accelerating the FHE comparison operation in the BGV scheme on a CPU/GPU system. BoostCom involves a multi-prong strategy including comparison-friendly parameter tuning, infrastructure acceleration (hybrid CPU/GPU paral-

lelization), algorithm-aware optimizations (slot compaction, non-blocking comparison semantic), as well as others. Together they provide more than an order of magnitude end-to-end speedups (11.1$\times$) over an industry-level FHE library running on a 16-core CPU.

CHAPTER 2: LITERATURE REVIEW

In this chapter, we categorize the literature based on the acceleration of the GPU-accelerated TEE system and the GPU-accelerated FHE system. In the first subsection, we narrate the related studies in confidentiality protection based on TEE. In the second subsection, we summarize the acceleration of FHE systems.

GPU Trusted Execution Environment (TEE)

There are a few recent proposals for TEE on GPUs. HIX [28] extended the Intel SGX interface to support the GPU enclave, which focuses on securing the GPU driver. The MMU design was also enhanced to prevent unauthorized access to the GPU memory-mapped I/O region. ZeroKernel [37] proposed a secure execution model that relies only on on-chip storage. This model assumes all of the kernel code can fit into the instruction cache and be stored there. It also assumes all of the PTE is cached in the TLB, then it removes all of the PTE from device memory and prevents page table reconstruction. These two proposals require no hardware changes to the GPU. Graviton [63] requires small hardware changes on the peripheral components. It assumes that the GPUs are using 3D stack memory, making it difficult to perform physical attacks. In Graviton, secure context isolation is achieved through an ownership tracking table. Using this table prevents unauthorized access to the victim address space. The focus of these three studies is to provide secure GPU execution without a lot of overheads, hence they did not include memory encryption.

Another approach to TEE is to bring CPU solutions to GPUs. The common counter scheme [45] uses counter-mode encryption and requires caches for the security metadata such as counter, MAC, Merkle Tree, and Common Counter Status Map. They observed that multiple counter values are

updated simultaneously, resulting in the same value for several counters. They proposed to group several data blocks to have a common counter to reduce counter cache misses. Yuan et al. [67] analyzed the performance implication of counter mode encryption for secure GPU memory. They observed that the increase in memory traffic due to accessing security metadata, including counters and MACs, is the main contributor to performance degradation. The memory traffic increase would affect GPUs using non-volatile memory more than those using DRAM due to the lower bandwidth capacity for larger memory space and crash-recoverable ability [6]. In subsequent work, Yuan et al. [68] proposed the PSSM scheme to reduce bandwidth overhead from the metadata. None of the above works address the inter-operability of CPU/GPU TEEs, hence they will still suffer the high performance overheads when data crosses encryption domains.

Table 2.1: Comparing LITE with prior GPU enclaves.

Aspect	Graviton [63]	Common Counter [45]	PSSM [68]	LITE
Memory encryption	No	Yes	Yes	**Yes**
Domain crossing	N/A	Yes	Yes	**No**
Hardware support	Low	High	High	**Low**
Flexible algo	N/A	No	No	**Yes**
Flexible app/data	N/A	No	No	**Yes**
Unified Memory	N/A	No	No	**Yes**
Deployability	Easy	Hard	Hard	**Easy**

Table 2.1 compares the solutions discussed above versus our software-based memory encryption LITE. Compared to the Common Counter and PSSM, LITE provides a unified encryption domain between CPU and GPU, allowing data to move from/to CPU TEE and GPU TEE without re-encryption. In contrast to Common Counter and PSSM, which require substantial hardware support (crypto engine per memory partition, metadata caches, etc.), LITE only requires small hardware support (write-once pages) for protecting kernel code. The flexibility of choosing an encryption algorithm is unique to LITE. Due to the flexibility of LITE, after establishing a common shared key, the GPU can use the same encryption scheme as the host CPU, leading to efficient

communication between them. For example, unified virtual memory (UVM) can be supported, and data can be moved back and forth between CPU and GPU without re-encryption. LITE also has the flexibility of applying memory security to select data and select applications, depending on the need at the host CPU, and requires no re-encryption. Finally, since it requires little hardware support, LITE can be deployed easily in current production GPU. In contrast, Common Counter and PSSM solutions require much more hardware support and are difficult to deploy.

Fully Homomorphic Encryption (FHE) Acceleration

Infrastructure acceleration is an approach to accelerate FHE operations with the use of hardware accelerators and efficient software implementation. It is used along with algorithmic improvement to achieve desirable performance. Table 2.2 shows the comparison of the prior works with Boost-Com on infrastructure acceleration of FHE. Among all the works on infrastructure accelerations, only ours focuses on boosting the latency of comparison operations on the BGV scheme. Furthermore, the infrastructure acceleration from the prior works could be divided into two categories: operation-wise acceleration and end-to-end acceleration. For the former, the acceleration is only targeting reducing the latency of each primitive FHE operation separately such as multiplication, addition, rotation, etc. Therefore they only estimate the total execution time of an application that runs on their proposal by the latency of each FHE operation. The proposals belonging to this category typically ignore the problem of dynamic memory allocation, different levels of the ciphertext operand, noise estimation, etc. since these problems may not arise when only accelerating each of the operations separately. For the latter, the acceleration is taking into account these problems and is typically used to accelerate the real-world library such as HElib[22] and Microsoft SEAL[44]. The end-to-end acceleration has a more immediate impact than operation-wise acceleration. It can be used to improve the execution time of the application that used the real-world HE library imme-

diately. In contrast, the operation-wise acceleration needs more work to gather them to be usable to truly run an application on it. Moreover, for both categories, the acceleration is divided by the type of hardware platform such as ASIC, FPGA, CPU (with new instructions), GPU, and mixed CPU/GPU.

Table 2.2: The comparison of BoostCom vs. prior works.

Name	Scheme	Comparison	End-to-End	Platform
SHARP[34]	CKKS	✓	✗	ASIC
CraterLake[52]	CKKS	✗	✗	ASIC
FxHENN[71]	CKKS	✗	✗	FPGA
TensorFHE[18]	CKKS	✗	✗	GPU
HE on GPU[46]	BFV	✗	✓	GPU
Intel HEXL[7]	BGV	✗	✓	CPU
BoostCom	**BGV**	✓	✓	**CPU/GPU**

Algorithmic acceleration. To boost the comparison operation on BGV/BFV, some algorithmic improvements have been proposed including, [25] and [60]. The scheme results in a slightly faster speed for performing comparison compared to TFHE when the number of messages being compared is large. However, when only comparing a single message in a ciphertext, the comparison latency becomes very expensive. Typically this problem arises when the comparison is used to determine the taken branch path. Our works proposed an optimization to mitigate this problem called *non-blocking comparison*. The optimization overlaps the comparison operation with other works that do not depend on the comparison result.

Operation-wise acceleration with GPUs. [18] and [64] proposes a GPGPU-based FHE acceleration solution called TensorFHE and HE-Booster, respectively. TensorFHE utilizes algorithm optimization, NTT optimization, and data layout optimization to achieve significant performance improvement for FHE arithmetic operations. This paper also utilizes tensor cores to speed up the NTT operation. HE-Booster improves the FHE arithmetic operation by improving the GPU NTT

implementation from [47] with fine-grain synchronization on every iteration of NTT computation.

Operation-wise acceleration with ASIC/FPGA. Several works in this category include [51, 36, 52, 71, 34, 35]. These proposals introduced an NTT unit for processing radix-2 NTT. CraterLake[52] is the first FHE accelerator to achieve high performance on unbounded FHE programs while prior accelerators are only efficient on a limited subset of simple FHE computations[51]. CraterLake[52] is a uniprocessor with specialized functional units that spans a wide vector space. The design is statically scheduled in order to take advantage of the regularity of FHE computations. SHARP[34], reduces the computation latency of the FHE operation by limiting the size of the prime modulus to only 36-bit. This will translate into lower memory bandwidth demand for the accelerator's memory thus improving the performance. Although the infrastructure acceleration with the use of ASIC/FPGA offers higher acceleration than GPUs, however, the lead time to the development of the hardware and the production is substantially long. Therefore, to get the benefit of this accelerator will need to wait for several years.

End-to-end acceleration. Intel HEXL introduced a new CPU instruction for processing 512-bit vectors in one go, speeding up element-wise operations and NTT. However, this only benefits power-of-two polynomial rings, and comparison operations remain slow in such rings. [46] proposed the acceleration of the BFV scheme in the Microsoft SEAL library for power-of-two polynomial rings, element-wise operation, and key-switching. Compared to our work, this work provides general FHE acceleration, while our proposal focuses on accelerating comparison operation on the BGV scheme.

CHAPTER 3: LITE: A LOW-COST PRACTICAL INTER-OPERABLE GPU TEE

[1] In this chapter, we address the challenges of accelerating confidential data computation protected with CPU TEE technology.

Introduction

Secure and private computation in the cloud is increasingly demanded by cloud computing users. To cater to that, chip manufacturers provide CPU TEE (Trusted Execution Environment), through which the processor provides a root of trust for guaranteeing confidentiality (and sometimes integrity) of computation and data against vulnerabilities in system software. A typical CPU TEE includes key features such as key management, attestation, and *memory security* [43] (memory encryption and/or integrity verification). Note that memory security is far costlier than others due to its continuous application during execution.

GPUs are increasingly widely used in the cloud. However, since current GPUs do not support TEEs, users sacrifice security and privacy when offloading computation to GPUs. Recently researchers have looked at providing TEE on GPUs [63, 28, 67, 68]. While these solutions work, they do not address inter-operability with CPU TEE. The CPU and GPU have their own encryption domains where each is the only one that can decrypt data it encrypted previously. Therefore, for a CPU to send data to a GPU, the data must be decrypted (by the CPU), re-encrypted in software, transmitted to GPU memory, decrypted in software, and then re-encrypted into the GPU encryption domain. Such *encryption domain crossing*, ignored in prior studies, incurs high costs.

[1]This work has been published in the 36th ACM International Conference on Supercomputing (ICS) 2022.

Figure 3.1 shows that the crossing overheads contribute to 60% higher execution time and up to 4.2× slowdown (322% overhead), which is much higher than 16% performance overhead from memory security alone. While these numbers were collected across different platforms, it is clear that encryption domain crossing dominates performance concerns.

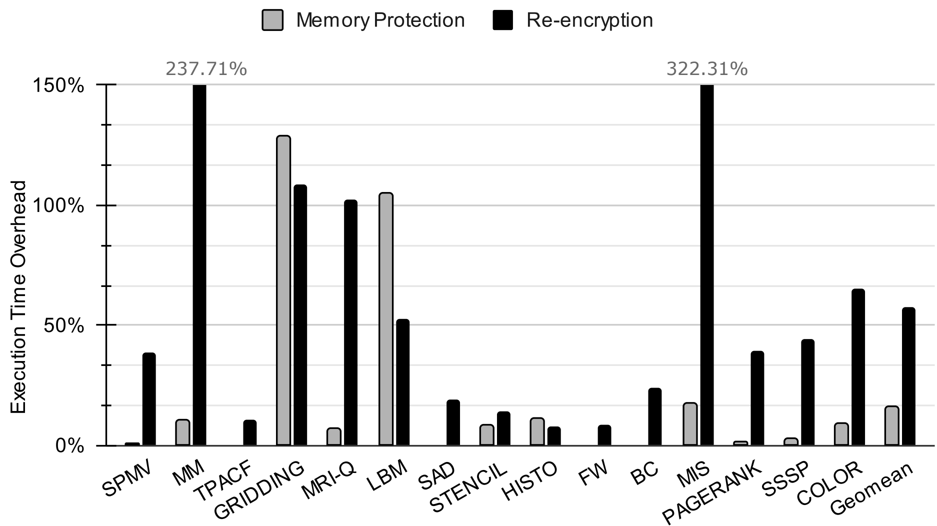

Figure 3.1: Execution time overheads of memory security with a state-of-the-art scheme (PSSM [68]) and domain crossing re-encryption overheads. The former was collected from GPU simulation with machine configuration from [68], while the latter is from a real machine described in Section 3.

Achieving inter-operability with CPU requires CPU/GPU TEE co-design, but co-designing is difficult for various reasons, e.g., GPU manufacturers may be different than CPU manufacturers, their design cycles may be different, etc. Furthermore, as an accelerator for the CPUs, the GPU TEE scheme should support a variety of GPU usage scenarios. For example, it may be paired with CPUs that vary in architectures, ISAs, types of CPU TEEs supported, and whether GPUs are paired with a single virtual machine (VM) vs. shared between VMs, etc. Given the wide variety of contexts in which GPUs may be deployed, it is important that GPU TEE support is as flexible as possible.

A flexible GPU TEE design provides additional benefits. Some workloads may process sensitive

data, hence confidentiality is required, but confidentiality may not be a priority for many others (e.g., graphics rendering or gaming). Furthermore, even for a single GPU kernel, in some cases, both input and output may be confidential, or only one of either input or output may be confidential.

However, the need for a flexible GPU TEE conflicts with the need to co-design GPU and CPU TEE. Co-designing them requires early coordination of CPU and GPU TEE design, which is difficult to achieve across companies. Furthermore, the wide variety of contexts and workloads in which GPUs could be used is hard to anticipate that early. Building many types of GPU TEEs to provide flexibility incurs a high fixed cost, e.g., supporting a single GPU memory security scheme already requires high die area overheads: one encryption engine for each memory partition, plus metadata caches to keep counters, MACs, and Merkle Tree nodes [68].

To achieve GPU/CPU TEE co-design without sacrificing flexibility, we propose software-based memory encryption, which we refer to as LITE. LITE requires only small hardware support for GPU TEE, and it relegates memory encryption to software. LITE achieves flexibility because software can choose different encryption algorithms to be in accordance with host TEE, selectively choose applications to apply encryption to, and select the subset of data to encrypt in an application, etc. LITE is possible in a GPU because GPU architecture supports explicit data movement (e.g., global to shared memory), unlike a CPU which relies only on caches that implicitly move data. LITE provides a library and APIs that the compiler/programmer can use to keep data encrypted in memory and only decrypt it before use.

To summarize, this paper makes the following contributions:

1. We propose a lightweight GPU TEE (LITE) solution that allows flexible CPU-GPU TEE co-design through the software layer.

2. We present three optimizations, masked shuffle, delayed shuffle, and selective padding, to

LITE that significantly improve its performance.

3. We show that despite relying on software for encryption, the optimized LITE incurs low performance overheads, with a geometric mean slowdown of 1.1% for regular applications. However, irregular workloads incur high performance overheads (55.7% on average). This overhead could be reduced by partial encryption to only 10.0% and 44.3% for input-only and output-only encryption, respectively.

Background

GPU Architecture and Unified Memory

A GPU consists of an array of Streaming Multiprocessors (SMs), and each SM has its own control unit, register file, L1 cache, and software-managed shared memory [48, 62]. Multiple SMs share an on-chip L2 cache with multiple banks to provide high L2 access bandwidth. One or more L2 banks are then connected to a memory controller to provide high bandwidth to access device memory. With discrete GPUs, data is typically moved between CPU main memory and GPU device memory over the PCI-e interface.

GPUs use Single-Instruction Multiple-Thread (SIMT) architecture to achieve high throughput. As a result, GPUs can tolerate/hide long latency by leveraging massive thread-level parallelism. However, they tend to be sensitive to bandwidth due to the high number of concurrent threads.

Prior to UVM [48], programmers had to manage memory explicitly by allocating memory in the host and device memory and moving data between the host and the device. A way to automate this is to use direct store to move data from CPUs to GPUs by exploiting data producer-consumer relationship [69] . With UVM, programmers can avoid explicit memory management and rely on

on-demand paging managed by UVM. UVM enables CPU and GPU to share the virtual memory space. Programs executed on the GPU are no longer limited by the size of device memory and can instead access host physical memory through a GPU virtual address.

Host-side TEE

A TEE may be designed to protect an application, a system, or memory. For the former, a ring-3 secure execution environment for code and data is provided by hardware (e.g., Intel SGX enclave) to protect against system software (OS and hypervisor) vulnerabilities as well as other code portions of the application that run outside the enclave. For the latter, a ring-0 secure execution environment is provided by hardware to protect a system (OS and applications) from vulnerabilities in the hypervisor or other systems. Examples include AMD Secure Encrypted Virtualization (SEV) [31] and Intel Multi-Key Total Memory Encryption (MKTME) [27]. AMD SEV and Intel MKTME are geared toward virtualized cloud computing servers where multiple virtual machines may share the same physical server. Each VM is provided a unique hardware ID with associated unique keys. Finally, the memory may be protected with encryption without regard to the software environment, such as in Intel Total Memory Encryption (TME) [27] and AMD Secure Memory Encryption (SME) [31]. TME is especially important for non-volatile memory with data remanence problems. A common support across all three types of execution environments is memory encryption. However, the type of memory encryption differs based on the goal of the memory security protection.

User-level enclaves such as SGX rely on counter-mode encryption and integrity verification relying on MACs and integrity tree [16]. Counter mode encryption is vulnerable to revealing plaintext of data if the counter can be changed or replayed by the adversary [66]. Thus, MACs and the integrity tree are an integral and necessary part of guaranteeing confidentiality in counter mode en-

cryption, in addition to detecting integrity violations. In contrast, XTS mode encryption (shown in Figure 3.2) is often deployed without integrity verification as data confidentiality is not dependent on integrity verification. Thus, since the XTS encryption mode is not prone to revealing plaintext, integrity verification is only needed for detecting integrity violations. Without integrity verification, the attacker may modify the ciphertext of data in memory without being detected; however, the tampered ciphertext will be decrypted to an unpredictable plaintext value, which may be of little value to the attacker. Since Intel SGX usage is currently limited to a small portion of an application that is highly security sensitive, and our goal is to provide a TEE for a whole application or system, we instead focus on non-counter mode memory encryption.

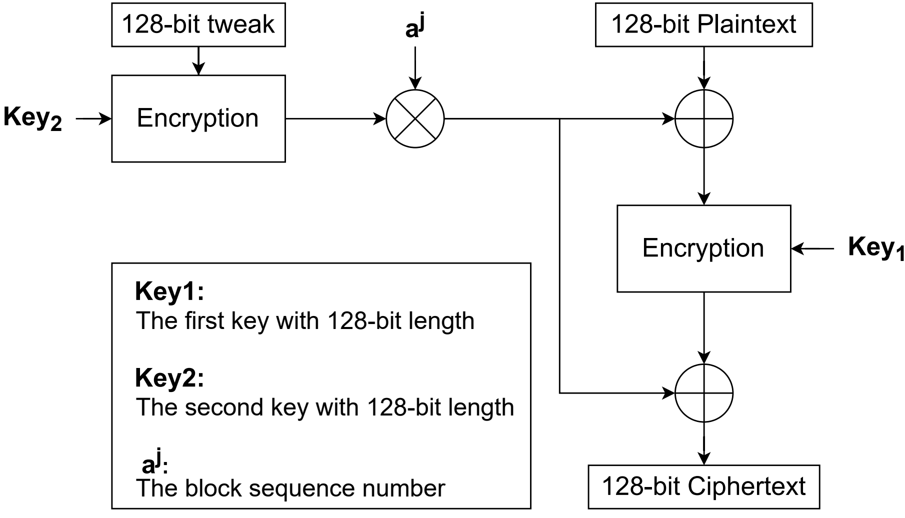

Figure 3.2: The block diagram of the AES-XTS encryption mode used in Intel MKTME.

Intel Total Memory Encryption (TME) [27] allows for the encryption of the entire physical memory of a system using the AES-XTS algorithm with a 128-bit key. The encryption key is generated using a hardware random number generator residing in the System-on-Chip (SoC). Multi-Key Total Memory Encryption (MKTME) [27] extends the TME to support multiple keys, and each page in the physical memory can be associated with a key. A process or the OS can read the plaintext of a page only when it has the right key in its page table entry for the page. The SoC

supports a fixed number of encryption keys, and software can use the keys to encrypt any page in the memory.

Similarly, AMD Secure Memory Encryption (SME) [31] provides main memory encryption using a single key generated by the AMD Secure Processor (AMD-SP). Encryption is performed by AES encryption engines located in on-die memory controllers using a 128-bit key. AMD Secure Encrypted Virtualization (SEV) [31] extends SME by providing cryptographic isolation for a VM from the hypervisor and other VMs. This isolation is achieved by assigning a hardware tag and key to each VM and tagging pages for that VM. This assures that the plaintext of the VM pages to only be readable by the VM itself.

Figure 3.2 illustrates the AES-XTS with 128-bit keys and a tweak to encrypt a 128-bit plaintext. The tweak represents the address of the data being encrypted or decrypted [24] to ensure the same plaintext value at different addresses results in different ciphertexts. Compared to counter-mode encryption, the AES-XTS mode does not require counters to be kept to perform its operations, hence eliminating the counter integrity problem. Thus, other metadata (MACs and integrity tree) are also no longer needed to ensure confidentiality.

Threat and Trust Model, Scope of Work

Both Intel TME and AMD SEV assume a threat model where the attacker has both a software and physical attack surface. The software attack surface is defined by the attacker having control over privileged software such as the OS and hypervisor. The physical attack surface is defined by the attacker having limited physical access to the machine to employ passive physical attacks such as snooping or scanning attacks but cannot modify data stored in memory. To be compatible with the host-side TEE, we assume the same attack model for GPUs.

We consider the following attacks to be out of the scope of this paper: active physical attacks (i.e., where data in memory is physically tampered with), side-channel attacks, and availability attacks.

Our trust model is as follows. We assume that the CPU, GPU, and CPU/GPU memories are trustworthy components in the sense that they operate correctly according to their specifications, free of design errors, faults, and trojan circuits. We assume that the system already has secure key storage in place, where the CPU and GPU have built-in public/private cryptographic key pair, with the public key readable from chip pins and the private key stored securely in a non-volatile manner in the CPU and GPU chips. Both chips are also assumed to have non-volatile storage to keep other keys, including session keys. Furthermore, we assume that a trusted party, such as the system integrator or the cloud administrator, has provided GPU public key to the CPU and CPU public key to the GPU. Alternatively, the CPU and GPU could automatically exchange public keys the first time they are connected. Thus, the CPU and GPU have a mechanism to initially trust each other.

Building on this trust, the CPU and GPU may initiate a Diffie-Hellman key exchange to establish a different shared session secret key, which enables a private communication channel between the CPU and GPU. If there is more than one CPU or GPU in the system, multiple secret keys must be tracked and stored on-chip. The shared secret key enables a private, authenticated communication channel that persists until the system shuts down. When the system is rebooted, the BIOS execution results in a new shared secret key to establish the communication channel.

Furthermore, our LITE scheme relies on a hardware feature where kernel code cannot be tampered with once it is loaded into GPU memory. Since active physical attacks are out of scope, the kernel code would not be altered by physical attacks. So, the possible threats would be malicious software, such as a driver. The code may be protected by the GPU page table, which sets the pages used for code as read-only.

The Design of LITE

In this section, we describe the design of LITE and discuss how, through a software solution with small hardware support, LITE allows GPU TEE to be co-designed with CPU TEE and provides great flexibility in which application or data is encrypted. Our discussion will start with rationale and overview, APIs, hardware support, and optimizations that enable LITE to achieve very low overheads for many applications.

Rationale and Overview

Figure 3.3 contrasts the typical data flow of current hardware-based GPU TEE design vs. our LITE. With current hardware GPU TEE (Figure 3.3(a)), because CPU and GPU TEEs use different encryption schemes and/or have their own encryption keys, they need to establish an intermediate ciphertext format that can be encrypted/decrypted by both the CPU and GPU. Data sent by the CPU must first be decrypted from the ciphertext C_1 in the CPU TEE domain and encrypted to the intermediate ciphertext form C_2 ①. After memory copy to device memory ②, the ciphertext needs to be re-encrypted again into the GPU TEE domain ③. When data is read by GPU, it is decrypted by the encryption engine ④ and stored in plaintext P in the on-chip GPU memory hierarchy. Notice the two sets of decryption and encryption that are added to the critical-path delay of CPU sending its data to GPU. This has to be repeated in the reverse direction as the GPU sends its computation result to the CPU at the end of kernel execution.

In contrast, in LITE (Figure 3.3(b)), if the GPU is in the same encryption domain as the CPU, we eliminate the re-encryption of data between CPU and GPU. Data can be directly copied from host memory to device memory ①. In theory, one could co-design GPU memory encryption to match that of the CPU and also avoid cross-domain re-encryption. However, this is difficult in practice for

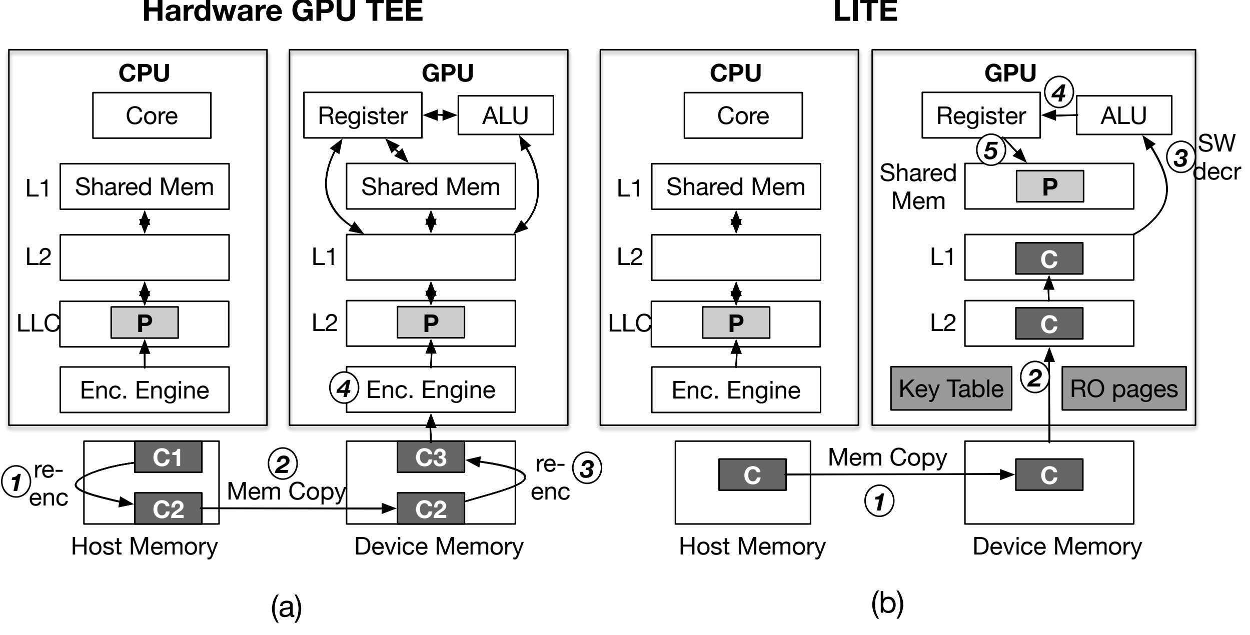

Figure 3.3: The LITE vs Hardware-based GPU TEE typical data flow from host to device.

several reasons. First, the GPU manufacturer may be different from the CPU manufacturer, making co-design difficult. Second, CPU may have different design and update cycles than GPU. For example, AMD Epyc 7251 processor uses XE-based encryption while AMD Epyc Embedded 3151 processor uses XEX-based encryption mode, although produced only eight months apart [65]. Finally, GPU may be used in many different scenarios to execute different workloads, hence ideally, GPU memory security should have a large degree of flexibility to match use cases and workloads. Therefore, in LITE, we focus on software memory encryption approach that can be deployed in GPU independently on CPU. A consequence of LITE's software approach is that data is brought into the GPU chip in ciphertext form ②. Then software decrypts data using ALU ③ operating on data on registers ④. Data may also be stored temporarily in plaintext in on-chip shared memory ⑤. LITE's software memory encryption approach is enabled in GPU because data movement into shared memory is controlled by software. The same approach is not deployable in CPU because

data movement in caches is transparent to software.

There are several inherent advantages to LITE beyond avoiding domain-crossing re-encryption overheads. It can readily support unified virtual memory (UVM) because a page can be copied between host and device memory without any ciphertext transformation. Second, data is stored in ciphertext form in GPU caches, which are shared by multiple SMs that may run different kernels concurrently. While we are not aware of current security attacks that leak cached data, if such an attack arises in the future due to GPU design bugs, only the ciphertext leaks out.

Challenges in achieving LITE are numerous, and we seek to address them in this paper. Implementing the same encryption algorithm in software in GPU is only the first step in sharing an encryption domain with CPU TEE. First, simple APIs need to be defined for kernels to use. Second, another necessary ingredient is to share the same encryption metadata as used in the CPU TEE, including any tweak inputs (e.g., host physical address). Third, there needs to be hardware support at GPU to ensure that encryption software cannot be tampered with by the attacker and that encryption keys can be stored securely in GPU chip. Fourth, software encryption performance bottlenecks are aplenty and need to be addressed, including high latencies, warp divergence, and the challenges in collecting 128-bit of data from multiple threads to be encrypted/decrypted with block cipher algorithms. We will discuss them in the rest of the section.

Encryption APIs

Since AES is block-based encryption, data is encrypted by the host in blocks of 128 bits (if 128-bit AES is used). To decrypt data correctly, the same block must be gathered and decrypted. If the access pattern of the GPU is to contiguous data elements, then the encryption block can be gathered simply by collecting data from the registers of neighbouring threads. To do that, we rely on *shfl*, which is a warp-level shuffling instruction that enables a thread to read the registers of other

threads. After decryption, data can be redistributed back to various threads by another round of shuffling. However, the if memory access pattern involves non-contiguous locations, assembling an encryption block is not as straightforward. In this case, we rely on padding so that each data item is expanded into a 128-bit single encryption block.

To ease adoption of LITE in the kernel code, we provide high-level AES APIs shown in Table 3.1. The first two calls are used for the case where adjacent threads in a warp access contiguous data, while the last two are provided when adjacent threads do not access contiguous data, hence padding is used (int_128 or a vector of four ints/floats) per thread.

Table 3.1: Encryption APIs

Interface to encryption and decryption
`encrypt(data, variable_addr, addr_type, enc_mode)`
`decrypt(variable_addr, addr_type, enc_mode)`
`encrypt_v4(data, variable_addr, addr_type, enc_mode)`
`decrypt_v4(variable_addr, addr_type, enc_mode)`

The decrypt/decrypt_v4 API loads the ciphertext and returns the plaintext. The encrypt/encrypt_v4 API takes the plaintext data in a register and returns the ciphertext generated using the key and the address tweak. In the last two arguments for each API function, the type of address (virtual or physical) and the encryption modes are specified to match those used by the CPU TEE. Listing 3.1 shows the implementation of APIs. `decrypt` shows the use of shuffling to assemble multiple contiguous 32-bit values in neighboring threads' registers into one 128-bit (Step 4), which is then decrypted (Step 5), and redistributed back to different threads' registers (Step 6). For `decrypt_v4` and `encrypt_v4`, with each thread accessing data using the float4 or int4 data types, data assembling/redistribution is not needed, hence AES encryption function is invoked directly (Step 4 and 3, respectively).

Listing 3.1: Implementation of AES APIs.

```
1  unsigned int decrypt(unsigned int v_addr, bool addr_type, int enc_mode) {
2      //step 1: configure encryption mode
3      AES_Encryption_Mode(enc_mode);
4      //step 2: set address tweak type
5      AES_Address_Type(addr_type);
6      //step 3: accessing global memory at v_addr
7      unsigned int temp = (unsigned int) (* v_addr);
8      int_128 buff;
9      unsigned int p_text //plaintext
10     //step 4: assemble 128-bit data block
11     if(tid % 4 == 0) { //tid is the thread id
12       buff[0] = temp; //temp from thread i, i is a multiple of 4
13       buff[1] = __shfl_down_sync(0xffffffff,temp, tid + 1) //temp from thread i+1
14       buff[2] = __shfl_down_sync(0xffffffff,temp, tid + 2) //temp from thread i+2
15       buff[3] = __shfl_down_sync(0xffffffff,temp, tid + 3) //temp from thread i+3
16       //step 5: decrypt data with AES_decrypt
17       buff = AES_decrypt(buff, v_addr);
18     }
19     //step 6: Distribute decrypted data to threads
20     if(tid% 4 == 0)
21       p_text = buff[0];
22     else
23       p_text = shfl_down_sync(0xffffffff, buff[tid%4], tid - tid%4]);
24     return p_text;
25 }
26 unsigned int encrypt(unsigned int data, unsigned int v_addr, bool addr_type, int
       enc_mode) {
27     unsigned int c_text; //ciphertext
28     int_128 buff;
29     //step 1: configure encryption mode
30     AES_Encryption_Mode(enc_mode);
31     //step 2: set address tweak type
32     AES_Address_Type(addr_type);
33     //step 3: assemble 128-bit data block
34     if(tid % 4 == 0) {
35       buff[0] = data; //data from thread i, i is a multiple of 4
36       buff[1] = __shfl_down_sync(0xffffffff,data, tid + 1) //data from thread i+1
37       buff[2] = __shfl_down_sync(0xffffffff,data, tid + 2) //data from thread i+2
```

```
      buff[3] = __shfl_down_sync(0xffffffff,data, tid + 3) //data from thread i+3
    //step 4: encrypt data with AES_encrypt
      buff = AES_encrypt(buff, v_addr);
    }
    //step 5: distribute encrypted data to  threads
    if(tid % 4 == 0)
      c_text = buff[0];
    else
      c_text = shfl_down_sync(0xffffffff, buff[tid%4], tid - tid%4]);
    return c_text;
}
int_128 decrypt_v4(unsigned int v_addr, bool addr_type, int enc_mode) {
    int_128 p_text //plaintext
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: accessing global memory at v_addr
    int_128 temp = (int_128) (* v_addr);
    //step 4: decrypt data with AES_decrypt
      p_text = AES_decrypt(temp, v_addr);
    return p_text;
}
int_128 encrypt_v4(int_128 data, int_128 v_addr, bool addr_type, int enc_mode) {
    int_128 c_text //ciphertext;
    //step 1: configure encryption mode
    AES_Encryption_Mode(enc_mode);
    //step 2: set address tweak type
    AES_Address_Type(addr_type);
    //step 3: encrypt data with AES_encrypt
    c_text = AES_encrypt(data, v_addr);
    return c_text;
}
```

AES Encryption and Address as the Tweak

As LITE encryption is based on software, the encryption algorithm and mode could be changed and updated to make it more secure, e.g., to add resistance to side-channel [40] or using a weaker algorithm such as DES [49] if higher performance is desired. For our implementation, we use the AES encryption from OpenSSL v1.1.1 and adapt it to CUDA [40]. It uses a T-table with a series of table lookups for each round and XORed to the round key.

To achieve CPU-GPU TEE interoperability, for host (CPU) memory encryption relying on a tweak, LITE needs to rely on the same tweak used by the host. Two approaches are possible. The first approach is to use the host physical address as the AES tweak. With this approach, the GPU needs to have the host physical address to decrypt data encrypted by the host and to encrypt the output for the host to decrypt. To achieve this, the GPU needs to keep the host physical address and uses it for decryption and encryption or be able to look it up. At least three techniques are possible. One technique is to add host physical address into the GPU page table such that given a virtual address, both host and GPU physical addresses can be looked up [59] [58]. With this technique, the GPU TLB also needs to be extended to include the host physical address. An alternative technique is for the GPU and host to share the same page table [38]. When the GPU misses in its TLB, the host IOMMU can be triggered to do a page table walk and provide the host physical address to the GPU for use in decryption and encryption. In this case, the GPU TLB can keep the host physical addresses but avoid modifications to the page table. Such a technique can utilize existing features such as NVIDIA Address Translation Service (ATS), available since Volta [61]. Finally, host physical addresses can be maintained in a separate data structure maintained entirely by the GPU. Such a structure needs to be populated prior to kernel launch, protected as read-only during kernel execution and looked up as needed by the GPU for encryption/decryption. In any case, in order for the host to decrypt the GPU computation result correctly, we need to reserve data pages in the host

physical memory until the GPU computation is completed. In other words, LITE requires that all UVM data have consistent host physical addresses. It can be achieved by reserving & pinning host memory when UVM is allocated with the cudaMallocManaged API (a host function). If a page is first accessed by the GPU, the host physical address is sent through the page fault handler without data migration. If a page from UVM is migrated from host memory to GPU device memory, it is not unmapped, similar to page pinning. For the data generated from GPU, such as stack frames, for which there are no corresponding host physical addresses, the GPU physical addresses can be used as the tweak. This practice would not affect CPU-GPU interoperability since GPU private data is not part of UVM and would not be accessed by the host.

The second approach is to use virtual addresses as the AES tweak. As UVM provides a unified virtual address space between host and GPU, the virtual addresses are the same on either side. Therefore, no additional changes are needed to achieve interoperability. Neither pinning nor modifications to paging are necessary. However, most host memory encryption uses physical addresses, which requires some changes to host memory encryption design, e.g., additional datapath to pass virtual addresses from the core to the memory controller. In our experiments on real GPU hardware, we assume this approach for the purpose of performance evaluation. The performance of the first approach (i.e., using host physical address as a tweak) would be very similar to the second approach if the GPU TLB is expanded to include the host physical address.

Kernel Code Adaptation

With our provided APIs, GPU kernel modification is quite straightforward. After determining the encryption algorithm, its mode, tweak, and address used in the tweak, global memory accesses to the data are examined. By default, we treat all GPU global memory as secure (encrypted) unless configured otherwise. If the memory accesses for four consecutive threads fall into contiguous

locations, then no padding is assumed, and the APIs to use are the first two in Table 3.1. Otherwise, padding is used, and the last two are used.

Kernel code can be adapted to use the APIs either manually or automatically by a compiler. With the compiler approach, the programmer can annotate the secure data variables through #pragma directives similar to OpenMP-style annotation, and the compiler transforms the directives into API calls. In this case, the compiler transformation forms a part of the trust base.

Listing 3.2 illustrates an example with tiled matrix multiplication. Global memory reads include `'a[row * n (i*tile_size+tx)]'` and `'b[(i * tile_size * n + ty* n) + col]'`. Since accesses are to contiguous 32-bit data, we simply convert them to `'decrypt(&a[row * n + (i* tile_size+tx)], addr_type, enc_mode)'` and `'decrypt( &b[(i * tile_size * n + ty* n) + col], addr_type, enc_mode)'`. Similarly, the global memory store statement `'c[(row*n) + col] = temp_val'` is converted to `'c[(row*n) + col] = encrypt(temp_val, &c[(row*n) + col], addr_type, enc_mode)'`.

Listing 3.2: Matrix multiplication kernel after code adaptation.

```
1  __global__ void tiledMatrixMul(float *a, float *b, float *c, int n,
2  int tile_size) {
3    __shared__ float A[SHMEM_SIZE];
4    __shared__ float B[SHMEM_SIZE];
5    int_128 buff1, buff2;
6    int row = by * tile_size + ty;
7    int col = bx * tile_size + tx;
8    int temp_val = 0;
9    bool addr_type = true; // true: phsyical, false: virtual
10   enc_mode = 1; // 0:ECB, 1:AES-XTS, 2:XE, 3:XEX etc.
11   // Sweep tiles over entire matrix
12   for (int i = 0; i < (n / tile_size); i++) {
13     A[(ty * tile_size) + tx] = (float) decrypt( &a[row * n + (i * tile_size + tx)],
       addr_type, enc_mode);
14     B[(ty * tile_size) + tx] = (float) decrypt( &b[(i * tile_size * n + ty * n) +
       col], addr_type, enc_mode);
15     __syncthreads();
```

```
16      for (int j = 0; j < tile_size; j++) {
17        temp_val += A[(ty * tile_size) + j] * B[(j * tile_size) + tx];
18      }
19      __syncthreads();
20    }
21    c[(row * n) + col] = encrypt(temp_val, &c[(row * n) + col], addr_type, enc_mode);
22  }
```

Listing 3.3: BFS kernel before and after code adaptation.

```
1    struct pad_128 {
2       int32_t data;     //32 bit
3       int32_t pad;      //32 bit
4       int64_t pad;      //64 bit
5   };
6   bfs_kernel(pad_128 *row, pad_128 *col, pad_128 *d, float_128 *rho, pad_128 *cont,
7              const pad_128 num_nodes, const pad_128 num_edges, const pad_128 dist)
8   {     ...
9   /*original code
10      for (int edge = start; edge < end; edge++) {
11          int w = col[edge];
12          if (d[w] < 0) {
13              (*cont) = 1;
14              ...
15          }  ... */
16  //code after padding
17      for (int edge = start; edge < end; edge++) {
18          int w = decrypt_v4(&col[edge], addr_type, enc_mode);
19          if (decrypt_v4(d[w], addr_type, enc_mode)<0){
20              int_128 i;
21              i.data = 1;
22              (*cont) = encrypt_v4(i, cont, addr_type, enc_mode);
23              ...
24          }
25  }
```

For non-contiguous access patterns, the data structure is padded before using the AES APIs to ensure that we would get 128-bit of data for encryption. The BFS function of the BC benchmark in

Listing 3.3 shows `'col[edge]'`, `'d[w]'`, and `'(*cont)'` in lines 11-13 not accessing contiguous locations, hence the 'int' type is converted to pad_128 type, which contains 96-bit padding beside the 32-bit data. Padding, however, increases the data structure size and incurs significant performance penalties.

Hardware Support

In order for LITE to work securely, some hardware support is needed. The hardware support required by LITE includes on-chip key storage, remote attestation, and code-integrity protection. Among them, on-chip key storage and remote attestation are supported in the latest NVIDIA Hopper GPU [5]. In LITE, code-integrity protection is achieved by page table protection, which sets the code pages as read-only. Such hardware overhead does not affect the performance of kernel execution. Note that compared to Hopper GPU, which enables encrypted CPU-GPU data transfer, LITE provides confidentiality of data stored in GPU memory.

Code Optimizations

Masked shuffle

As discussed earlier, we could use *shfl* instruction to collect 128-bit of data for decryption. *shfl* has an implicit thread synchronization that acts as a barrier to ensure threads in a warp have finished earlier computation prior to data exchange. This fact is important when considering applications with branch divergence. If threads in a warp have thread-divergent branches and execute a *shfl*, which is permitted in Volta and later GPU architectures [39], the shuffle is delayed until the threads with the longest branch path complete. This performance bottleneck is illustrated in an example in Figure 3.4 (top), where odd-ID threads diverge on a branch path than even-ID threads. When only

odd-ID threads need to load ciphertext from global memory, *shfl* forces them to wait for even-ID threads before data exchange in the decrypt function. To solve this, we propose a *masked shuffle* optimization that ensures only threads in the same branch path would be masked so that only those same threads will participate in data exchange. It takes advantage of the "mask" operand in the *shfl* instruction, which indicates specific threads in a warp that participate in the shuffling. We use the __ballot_sync() function to determine the set of participating threads in a warp for a branch path. This function is invoked before the diverging if statements. The result is used to set the mask of the shfl instruction such that only participating threads in a branch path perform the shuffle. In the example in Figure 3.4 (bottom), we use an odd mask in the decrypt function. By doing this, we let the odd ID threads complete the shuffle sooner, and the even threads do not participate in data exchange, thereby reducing the execution time.

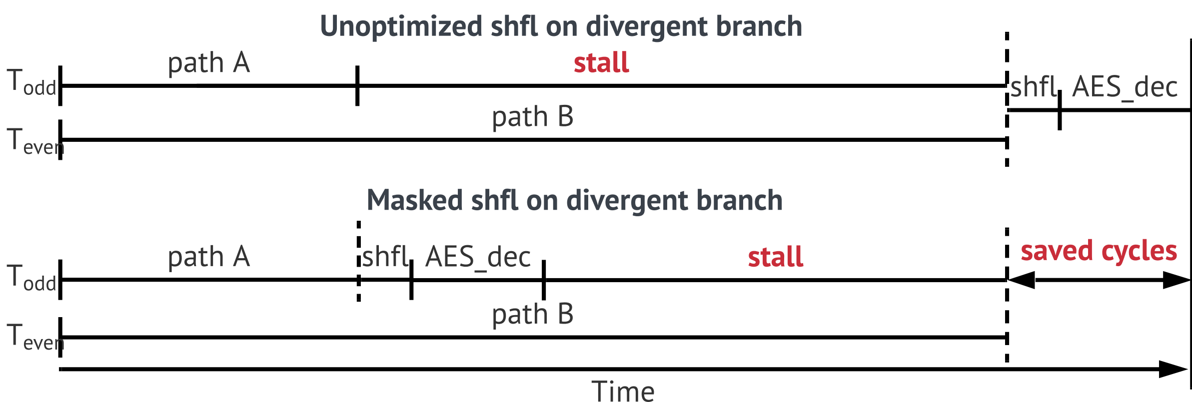

Figure 3.4: The timeline comparison between unoptimized shfl vs masked shfl. Following a divergent branch, only threads in path A fetch ciphertext from global memory.

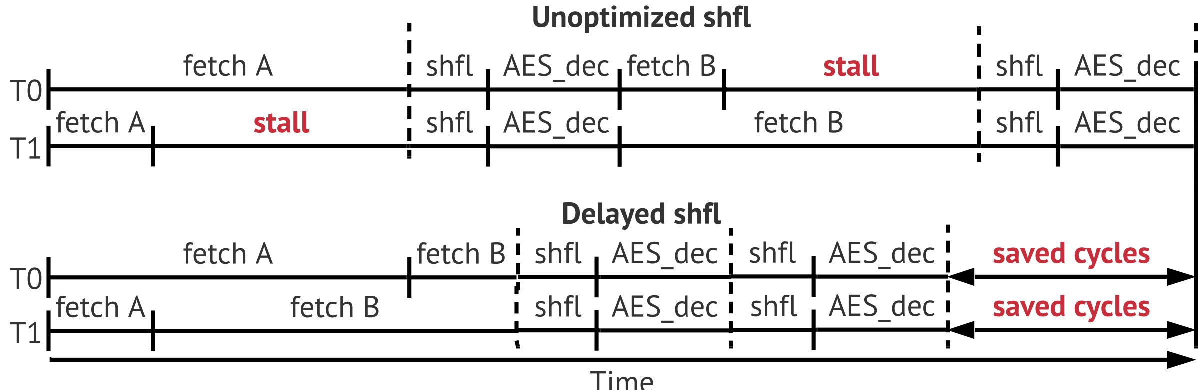

Figure 3.5: The timeline comparison between unoptimized code and optimized code with delayed shuffle. Fetch A and fetch B are two independent global memory accesses. Unoptimized code decrypts the data immediately while delayed shuffle decrypts the data after the data are loaded into shared memory/registers.

Delayed Shuffle

Another problem with the implicit synchronization of *shfl* is that it limits instruction overlapping. For example, lines 13 and 14 in Listing 2 are two independent global memory reads, 'fetch A' and 'fetch B', and they can be issued back to back. However, due to the shfl used in the decrypt function, fetch A and fetch B are sequentialized, as illustrated in Figure 3.5 (top). As a result, when there is memory divergence, i.e., some threads having cache hits while others in the same warp have cache misses, the longest latency is exposed. To address this problem, we propose a *delayed shuffle* optimization, which delays the decrypt function after independent global memory reads. In the example of Listing 2, the decrypt function is moved to right before the syncthreads function in line 15. In other words, the ciphertexts are first loaded in shared memory (both fetch A and fetch B), then the decrypt function is used to overwrite the shared memory array with plaintext. This would re-enable overlapping between fetch A and fetch B, as illustrated in Figure 3.5 (bottom),

leading to improved performance.

Selective Padding

If threads do not access contiguous data and the access pattern is hard to identify statically, there may be a data coherence problem that arises when two threads modify different data items in the same encryption block. We refer to this as an *encryption block false sharing* problem. Without hardware cache coherence, both threads will need to read the same encryption block in order to decrypt it, but this may create incoherent replicas. Earlier, we discussed that we eliminate the false sharing by padding the data structure such that each element is expanded to 128 bits. While this solution works, it increases memory footprint and bandwidth pressure. To improve on this, we note that having encryption block replicas only leads to coherence issues only if the block is modified. Thus, we perform *selective padding* optimization by differentiating the data access type; and skipping padding if data is read-only. The read-only attribute can be obtained from the programming model, e.g., the input buffers in OpenCL, compiler analysis, or programmer annotation.

Methodology

We evaluate LITE on an NVIDIA RTX 2080 GPU. The system runs on Ubuntu OS version 18.04 with NVIDIA driver version 440.33.01. For compilation, we use the CUDA version 11.0 and GCC version 7.1.0. We also use NVIDIA Nsight Compute to collect the hardware performance statistics. Each experiment was repeated 100 times, and we use the average of them.

To test LITE, we use a wide range of applications from two benchmark suites, Parboil and Pannotia, as shown in Table 3.2. They consist of both regular GPU and irregular GPU code. Parboil

benchmarks have regular GPU code and work well with GPUs since their memory accesses can be coalesced, and a 128-bit block of data is accessed either by one or four adjacent threads. Pannotia has irregular GPU code that accesses memory locations depending on the input, i.e., the sequence of memory accesses are randomly determined by the input. Irregular GPU codes have relatively poor performance on GPUs. As discussed in the previous section, we use padding for the irregular benchmarks, in which a 128-bit data block is not accessed by a single thread or by four consecutive threads in a warp.

Table 3.2: Benchmarks

Name	Input	Suites	Reg	Size(MB)	Bottleneck
SPMV	1138_bus.mtx	Parboil[57]	Yes	0.04	Memory
MM	medium	Parboil	Yes	55.8	Compute
TPACF	small	Parboil	Yes	0.88	Compute
GRIDDING	small.uks	Parboil	Yes	63.7	Compute
MRI-Q	32_32_32_dataset	Parboil	Yes	0.44	Compute
LBM	short	Parboil	Yes	2.2	Memory
SAD	large	Parboil	Yes	8.2	Memory
STENCIL	128x128x32	Parboil	Yes	2.1	Memory
HISTO	large	Parboil	Yes	4.1	Memory
FW	256_16384.gr	Pannotia[12]	No	0.19	Memory
BC	1k_128k.gr	Pannotia	No	1.7	Memory
MIS	ecology1.gr	Pannotia	No	28.5	Memory
PAGERANK	coAuthorsDBLP.gr	Pannotia	No	12.6	Memory
SSSP	NY.gr	Pannotia	No	14.4	Memory
COLOR	ecology1.gr	Pannotia	No	28.5	Memory

In Table 3.2, the 'Reg' column indicates whether the benchmark is regular or irregular, 'Size' indicates the input size in megabytes, and 'Bottleneck' indicates the performance bottleneck of the benchmark is compute or memory. The bottleneck categorization is determined mainly from the DRAM utilization and Streaming Multiprocessor (SM) utilization, as shown in Figure 3.6. If the DRAM utilization is higher than the SM utilization, the benchmark is categorized as memory bound, and vice versa. An exception is SAD; this benchmark is categorized as memory bound

since it has 98% utilization of the internal caches.

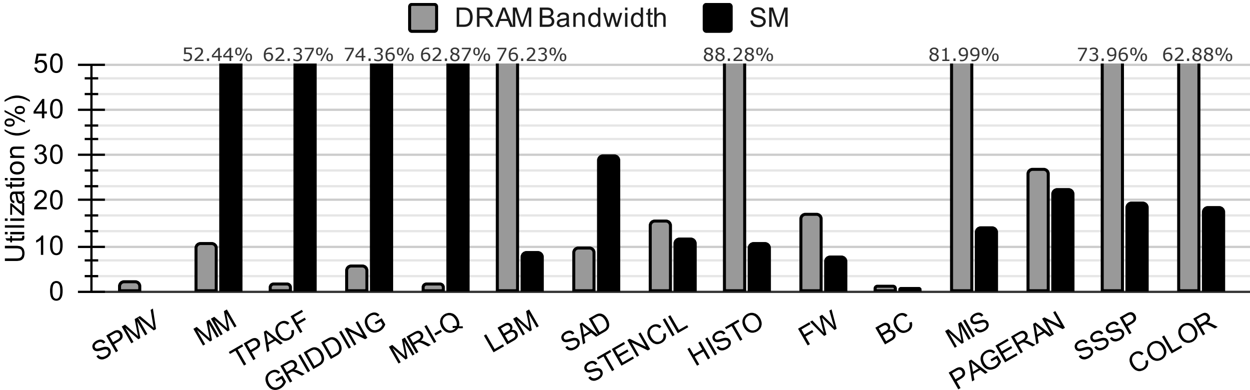

Figure 3.6: Streaming Multiprocessor (SM) utilization and DRAM bandwidth utilization.

Evaluation

LITE Performance Overhead

We present the kernel execution time overheads incurred by naive vs. optimized LITE implementations over the unsecured GPU baseline in Figure 3.7 and Figure 3.8 for regular and irregular code, respectively. The naive implementation always encrypts or decrypts data immediately after loading it or immediately before storing it to global memory. The optimized LITE applies all the optimizations described in Subsection 3. The execution time overhead is broken down into the time to perform encryption/decryption (*encryption*), the time to perform data shuffling without optimizations (*shfl-naive*), and the time to perform data shuffling with optimizations (*shfl-opt*). The last set of bars shows the geometric mean overheads across all benchmarks in the respective category. Note that the y-axes of the figure is capped at 3% for easier reading, but the actual magnitude overheads are shown in numbers for any bars that exceed the cap.

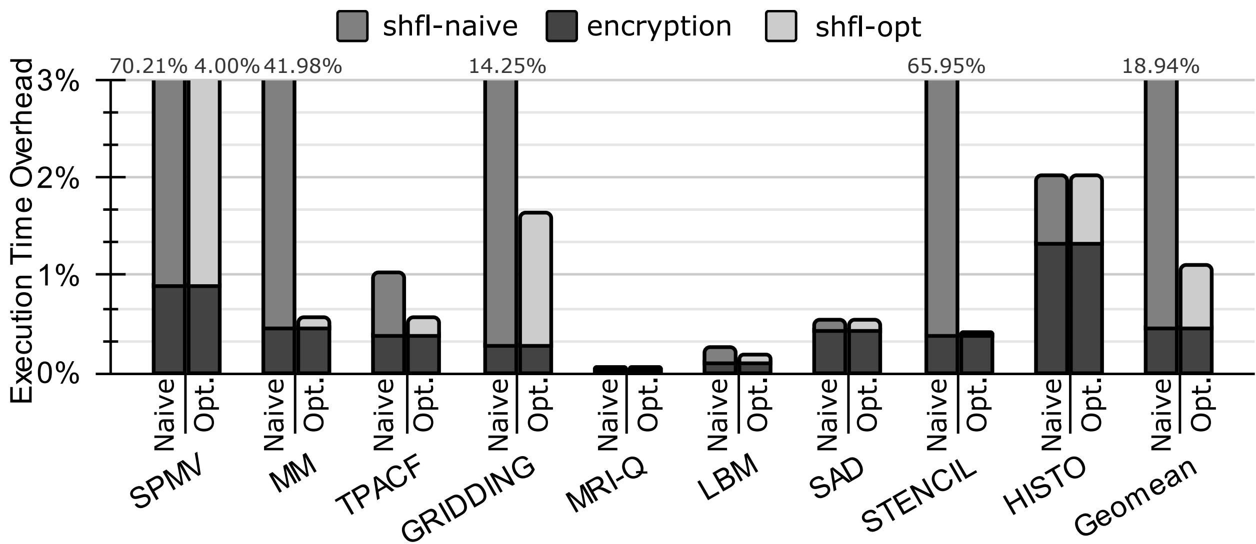

Figure 3.7: Kernel execution time overheads of naive LITE (shfl-naive+encryption) vs. optimized LITE (shfl-opt+encryption) over unsecured GPU baseline on regular GPU benchmarks.

For regular applications in Figure 3.7, the naive implementation of LITE incurs 18.9% mean overheads.The overheads are especially very high for four benchmarks: SPMV (70.2%), MM (42.0%), GRIDDING (14.3%), and STENCIL (66.0%). In contrast, our optimizations are extremely effective, bringing the geometric mean down to more than one order of magnitude smaller, at 1.1%.

Examining the source of the overhead, we note that the encryption time itself causes only 0.5% overhead in general. Although encryption/decryption latency is relatively high when performed in software, GPU is good at hiding it via thread-level parallelism, resulting in nearly negligible encryption overheads. This observation is also consistent with the findings in the prior work [67]. However, anything that reduces the degree of thread-level parallelism can easily introduce high execution time overheads. In particular, the *shfl* instruction exposes the load imbalance of path divergence between threads in a wrap, forcing the wrap synchronization to wait for the slowest execution path to be completed. Also, it limits overlapping among independent instructions. The

naive LITE shows that nearly all of the overheads for SPMV, MM, GRIDDING, and STENCIL, come from shuffling.

To verify if the implicit wrap synchronization in *shfl* is the culprit, we replaced the *shfl* instruction with *__syncwarp()* and noticed roughly the same overheads. We will now discuss the impact of optimizations on each benchmark.

For SPMV, we apply the masked shuffle optimization, ensuring that only threads that fetched data would participate in the corresponding data exchange and would be indicated active in the "mask". This optimization lowers the performance overhead from 70.2% to just 4.0%. The reason for the masked shuffle effectiveness is that SPMV may make some threads idle if their global thread id is beyond the length of the vector input. Thus, the idle threads should not be indicated as active in the "mask." We did not apply *delayed shuffle* because there are no independent data accesses.

For MM, we apply both the masked shuffle and delayed shuffle optimizations. We also replace some of the *shfl* with data fetch from the neighbouring addresses to collect the 128-bit encryption block. These optimizations significantly lower the overheads from 42.0% to just 0.6%, which is nearly two orders of magnitude improvement.

For GRIDDING and STENCIL, we applied both the masked shuffle and delayed shuffle optimizations. After applying the masked shuffle, the overhead decreases to 10.4% and 2.0% respectively, which is still somewhat high. However, after applying the delayed shuffle, the overheads go down to just 1.7% and 0.4% for GRIDDING and STENCIL, respectively. For TPACF, the masked shuffle optimization was applied similar to SPMV.

For irregular benchmarks (Figure 3.8), padding removes the false sharing coherence problem but significantly contributes to the overheads, incurring 206.5% geometric mean overheads for naive LITE due to increased working set size leading to higher bandwidth pressure. When we apply

selective padding to the benchmarks, the execution time overheads decrease substantially as bandwidth pressure decreases. The geometric mean overhead decreases to 55.7%.

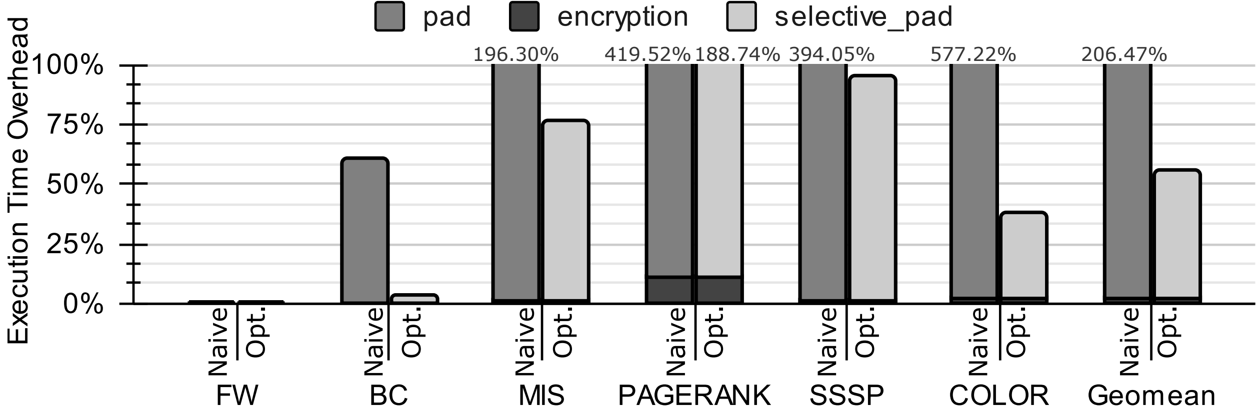

Figure 3.8: Kernel execution time overhead of naive LITE implementation (pad+encryption) vs. optimized LITE (selective_pad+encryption) over unsecured GPU baseline on irregular GPU workloads.

Only FW does not suffer from much overhead. The anomaly of FW is because its working set (Table 3.2) fits entirely in the L2 cache even after padding, hence padding does not increase bandwidth pressure. For other benchmarks, bandwidth pressure increases in padding due to the higher total number of memory accesses. In addition, miss rates often increase also. Figure 3.9 shows the L2 cache miss rate of unsecured GPU, full padding, and selective padding. Generally, the figure shows that padding increases the L2 miss rates, which are then reduced by selective padding. The only exception is COLOR. However, even though COLOR's L2 miss rate decreases with padding, its L1 cache miss rate increases (from 71% to 79%).

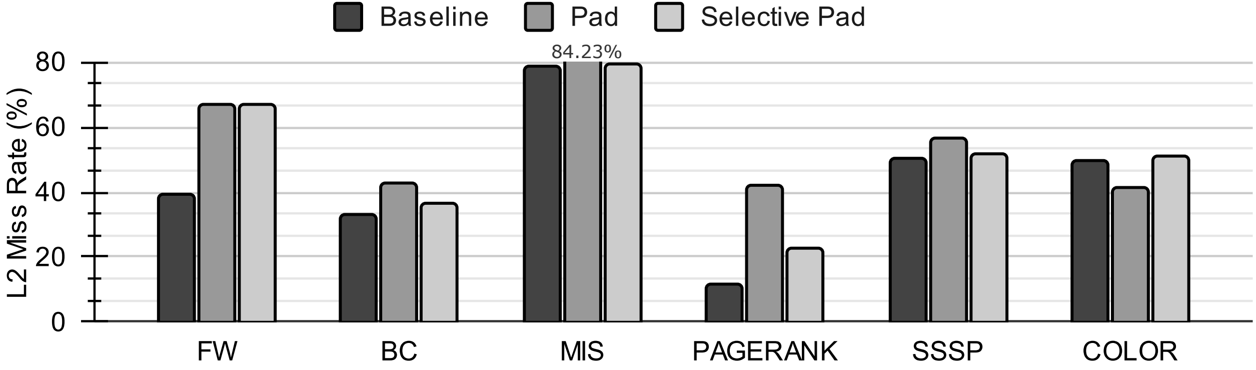

Figure 3.9: Comparison of the L2 miss rate of unsecure GPU (baseline) vs. full padding (pad) vs. selective padding (selective_pad) on irregular GPU code.

Benefit of Partial Encryption on LITE

Due to its software approach, LITE has great flexibility. We demonstrate one particular usage that stems from the flexibility, which is partial encryption. Here we explore scenarios where only the input or the output to GPU is confidential, hence needs encryption. Figure 3.10 shows the execution time overheads, normalized to full encryption (i.e., full encryption overheads are 100%), for input-only encryption with padding applied only to input, and output-only encryption with padding only applied to the output. Only irregular benchmarks are shown in the figure since full encryption overheads for regular workloads are already very small. The figure shows that partial encryption is generally effective, more so for input-only (82.0% lower) than for output-only (20.5% lower). Furthermore, its effect varies significantly across benchmarks, with some achieving nearly negligible overheads for input-only encryption (MS, PAGERANK, and COLOR) while others for output-only encryption (BC).

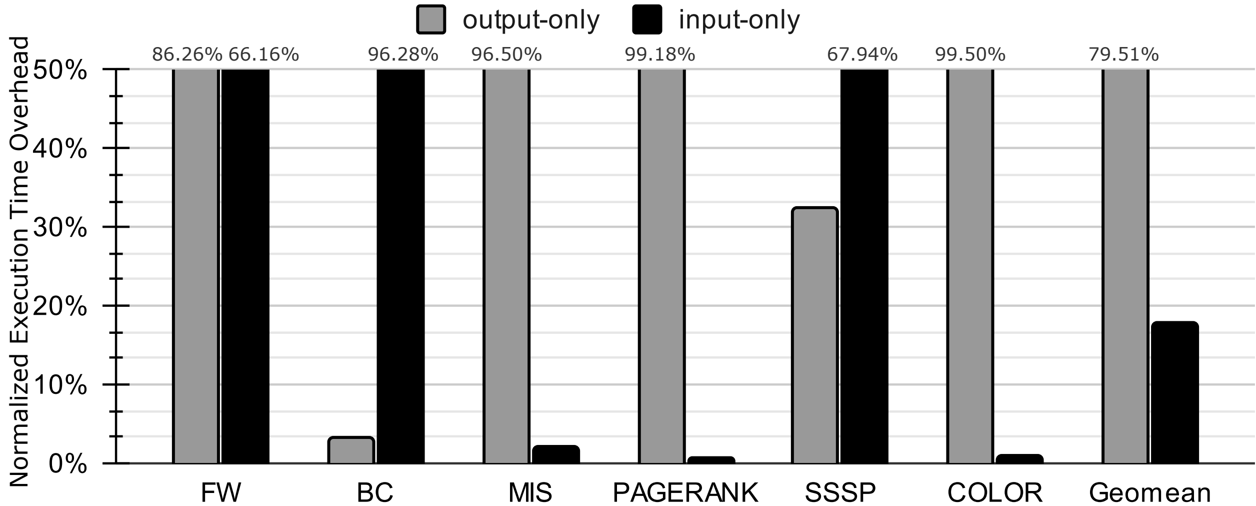

Figure 3.10: Normalized execution time overhead over full encryption of optimized LITE for input-only encryption vs output-only encryption on irregular GPU code.

Performance of Software Implementation of AES

As LITE does not use a crypto engine to encrypt the data, it relies on software implementation of AES to perform encryption. The software performance of AES encryption of a single round of AES takes up to 1011 cycles before the cache warms up, while after that, it takes only 34 cycles for a single round. Similarly, the overall AES encryption latency is 7880 cycles before the cache warms up, while after that, it takes 340 cycles. The latency is measured by taking the difference between the clock cycle before and after each round or the encryption function. After the cache warms up, the GPU only needs to perform bit operations such as XOR, AND, and bit shift for encryption. While before the cache warms up, the GPU needs to fetch the T-table into its caches followed by bit operations for performing encryption, thereby incurring higher latency.

We modeled LITE in GPGPU-sim [32] and compared it with PSSM [68]. We also simulated the re-encryption process by running a re-encryption kernel for the input of each benchmark on the simulator. We simulated on all benchmarks listed in Table 3.2 up to 1B instructions or finished earlier. The results are shown in the Figure 3.11. As shown in the figure, for regular codes, LITE always achieved lower performance overhead compared to PSSM. For irregular codes, PSSM has lower performance overhead if the re-encryption overhead is not included. With re-encryption, PSSM shows a similar overall performance to LITE. Across all benchmarks, LITE achieved 11.7% performance overhead while PSSM with re-encryption achieved 68.8% overhead. From this, we could see that a key advantage of LITE is its interoperability, which eliminates the high re-encryption cost.

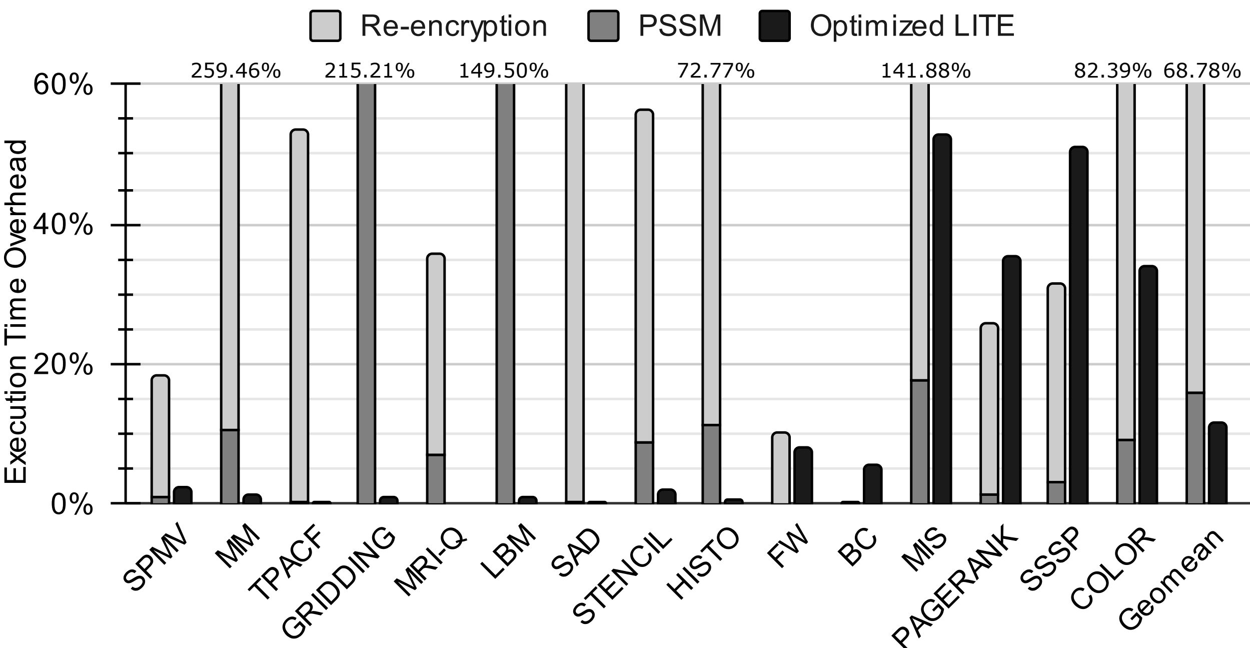

Figure 3.11: Comparison of execution time overhead of LITE vs. PSSM with re-encryption overhead evaluated in GPGPU-sim.

Conclusion

In this work, we proposed a software-based TEE for GPUs (LITE). We observed that when CPU and GPU TEE are not co-designed, communication between them incurs high performance overheads because of encryption domain crossing. Since LITE is software-based encryption, its encryption scheme can be co-designed to match host-side TEE, even after tape out. LITE only needs minor architecture support. Measured on NVIDIA RTX 2080 GPU, naive LITE implementation incurs substantial performance overheads. We proposed three different optimizations including masked shuffle, delayed shuffle, and selective padding. Together, these optimizations are effective. LITE incurs execution time overheads of only 1.1% and 56% for regular and irregular benchmarks, respectively.

CHAPTER 4: BOOSTCOM: BOOSTING THE COMPARISON OPERATION ON ENCRYPTED DATA USING GPU

In this chapter, the GPU is used to accelerate the FHE implementation inside the HElib library.

Introduction

Data privacy in cloud computing requires technologies such as Trusted Execution Environment (TEE) or Fully Homomorphic Encryption (FHE) [20]. FHE is a cryptographic technique that enables arbitrary computational operations directly on encrypted data, without ever seeing the plaintext. In contrast, TEE computation processes plaintext within the CPU boundary, despite being cryptographically segregated from the system software and external to the CPU. Operating on plaintext produces plaintext-dependent architecture behavior that creates various side channels that have been proved difficult to close entirely.

There has been a surge of interest from the industry in FHE acceleration recently [44, 3, 7] as FHE may play a pivotal role in facilitating computation on private data in the cloud without disclosing its plaintext. FHE has been cited to be applicable for many types of computation, including machine learning, and big data analytics, on various application domains that include healthcare, finance, genomics research, secure voting systems, and private information retrieval, where it helps maintain stringent privacy regulations [10, 17, 26, 70]. Figure 4.1 illustrates the FHE workflow, which includes client-side encoding, encryption, server-side computation, and subsequent client-end decryption and decoding, which assures data confidentiality even on potentially untrusted servers.

Various FHE schemes, including BGV [9], CKKS [13], and TFHE [15], have emerged in the previous decade. These schemes exhibit distinct characteristics and operational compatibility. TFHE

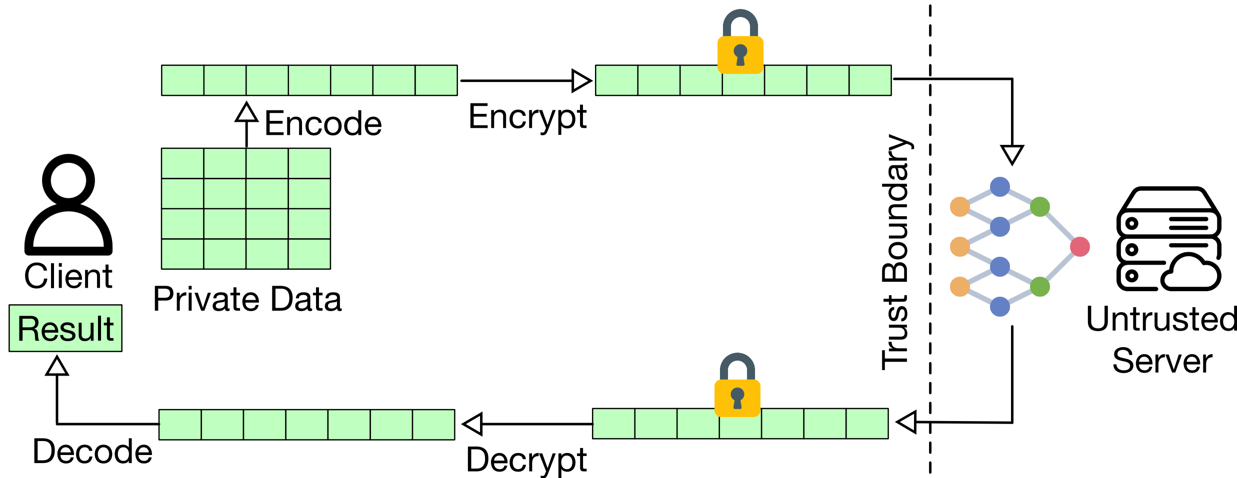

Figure 4.1: The workflow of performing computations on encrypted data that is transferred to an untrusted server using FHE.

scheme, for instance, is geared towards bitwise operations, whereas CKKS primarily operates on approximate floating-point computations and BGV specializes in integer arithmetic operations. However, it is noteworthy to mention that CKKS, in its pursuit of approximate computing, undergoes a reduction in precision as every operation it conducts impacts the ciphertext's fractional value [29]. Moreover, the comparison involving an approximated polynomial in the CKKS framework leads to non-negligible errors [14] whereas comparisons in BGV introduce no errors and exhibit faster execution times [25]. TFHE, though proficient in bitwise comparison, exhibits a slower performance in arithmetic operations on integers. Conversely, BGV not only facilitates integer arithmetic operations natively but also supports batched data processing and exact comparison. Therefore, BGV appears promising for the efficient implementation of FHE applications that heavily rely on integer arithmetic operations and exact comparisons.

Nevertheless, the BGV scheme is not without its limitations, particularly the sluggish and complex comparison operation. To execute a single comparison, it requires $3p-5$ non-scalar multiplica-

tions in conjunction with several additions, rotations, and scalar multiplications, with p denoting the plaintext modulus [60]. A variety of applications, encompassing scientific computations and machine learning workloads, depend heavily on this comparison operation. These applications necessitate the comparison of two elements within the encrypted data, subsequently yielding results whether they are equivalent, less than, or greater than.

Recognizing that a comparison operation may create a performance bottleneck in BGV, there has been an effort to rely on an algorithmic approach to accelerate it [25]. The algorithmic approach reduces the comparison complexity to $2p-6$ (Bivariate case) and $\sqrt{p-3}+\mathcal{O}(\log p)$ (Univariate case). Although an algorithmic approach is valuable, we are of the view that it alone may not be adequate to meet the requirements of high performance. Proposals have been made to switch between FHE schemes like TFHE-BGV [8] and TFHE-CKKS [41]. However, these transitions are still costly, with over 70x latency compared to BGV [25]. Therefore, in this paper, we propose an infrastructure acceleration approach, which we call *BoostCom*, where we offload comparison to the Graphics Processing Unit (GPU) and apply various optimizations. We note that infrastructure acceleration approach has been pursued successfully for various other operations such as encryption, decryption, multiplication, bootstrapping, and other power-of-two polynomial ring operations [64, 30, 46, 53, 18], including on FPGA [50, 56], and ASIC [51, 52, 36]. However, they have all neglected the comparison operation, which is the focus of this paper.

In this work, we propose an infrastructure acceleration approach for accelerating the comparison operation. A single comparison may be up to multiple orders of magnitude slower than multiplication. Hence, accelerating comparison is challenging, requiring us to use a multi-prong strategy that includes comparison-friendly parameter tuning, hybrid CPU/GPU parallelization, slot compaction, non-blocking comparison semantics, branch removal, and layout optimization, as detailed below.

First, the choice of the polynomial ring significantly impacts the overall performance of BGV. Prior

research presumed the use of a power-of-two polynomial ring. However, these studies overlooked the fact that such a ring leads to a considerably slower comparison operation, as they did not delve into the aspect of comparison. When considering the comparison operation, it becomes evident that a non-power-of-two polynomial ring holds a substantial performance advantage for the BGV scheme, offering a smaller ciphertext size and a larger number of slots while maintaining the same security level as the power-of-two ring. Furthermore, the comparison operation exhibits a substantial difference in execution time on different rings. For instance, a single comparison may require several hours with a ring of 16,384, contrasting with just a few seconds for a ring of 18,157.

Second, we propose hybrid parallelization where the multicore CPU exploits parallelization at a higher level (digit-level), while the GPU exploits parallelization of primitive FHE operations at a lower (library) level. This enables both CPU and GPU to work cooperatively to improve BGV comparison performance.

Third, we observed that in machine learning workloads, comparison occurs after other operations, such as matrix multiplication or convolution. This creates an opportunity to perform *slot compaction* prior to performing the comparison. We introduce a slot manager that keeps track of slot utilization inside a ciphertext and performs compaction in order to minimize the number of slots used, resulting in lower memory usage and performance improvement.

Fourth, we propose *non-blocking semantic* for comparison that allows the overlap of comparison with other computations. The semantic allows comparison to be executed on another CPU thread while the main CPU thread continues executing the next code segment concurrently until the main thread needs to use the result of the comparison. To increase the distance until the use of the result, we perform code straightlining.

Finally, we accelerate each primitive FHE operation on GPU, targeting key performance-intensive codes on HElib [22] such as BluesteinNTT, BluesteinFFT, and element-wise operation. We apply

optimizations that include branch removal, plan reuse, and layout optimization to achieve better spatial locality.

We implemented the optimizations on a real-world library (HElib) which enables us to evaluate end-to-end performance (instead of operation-wise evaluation in many prior studies) reliably. We evaluate several applications including sorting, finding minimum elements, multi-layer perceptron (MLP), image re-colorizing, and a private query. Our evaluation shows that the proposed acceleration is effective in boosting the performance of the comparison and the application that use it. Across the five benchmarks, it achieves *end-to-end* geometric mean speedup of 11.1$\times$ (even up to 26.7$\times$), over an industry-standard FHE library running on 16-core CPUs.

To summarize, this paper makes the following contributions:

1. We proposed an infrastructure acceleration approach for comparison in the BGV FHE scheme.
2. We proposed a multi-prong strategy for accelerating comparison, including comparison-friendly parameter tuning, hybrid CPU/GPU parallelization, slot compaction, non-blocking semantics, branch-removal, as well as other minor ones (plan reuse and layout optimization). These optimizations are wrapped in a new library cuHELIB, which extends HElib utilizing GPUs.
3. We conducted a thorough evaluation of our scheme, considering end-to-end measurements that include CPU-GPU memory copy, kernel launches, and synchronizations. This approach provides a more holistic assessment compared to extrapolating from operation-wise measurements. The evaluation covered five applications: Sorting, Finding Minimum Element, Multi-Layer Perceptron (MLP), Image Re-Colorizing, and Private Query Database.

BGV Scheme

The BGV scheme is a lattice-based encryption based on Ring Learning with Errors (RLWE) problem [9]. RLWE is a challenging mathematical problem in lattice-based encryption that creates a foundation for developing safe encryption schemes. Table 4.1 shows the essential BGV parameters. Key parameters include p, m, and N. p defines the plaintext modulus; its increase enlarges plaintext space but slows down comparisons. The roles of m and N will be outlined later.

Table 4.1: Parameters used in BGV and comparison operation.

Parameter	Description
p	Plaintext coefficient modulus.
m	The order of the cyclotomic ring.
N	The degree of the cyclotomic polynomial.
Q	The product of (prime) moduli: $Q = \prod_{i=0}^{L} q_i$.
L	Maximum (multiplicative) level.
λ	Security level of a given BGV instance.
ω	Root of unity of twiddle factor for NTT.
d	The dimension of a vector space over a finite field.
l	The length of vectors to be compared.

In BGV scheme, a plaintext is encoded into a polynomial and encrypted to form a ciphertext polynomial. Computation can be performed on the ciphertext, yielding a result also in ciphertext form, which requires decryption to obtain the plaintext. Both plaintext and ciphertext polynomials reside in the same ring with different coefficient moduli, where the ciphertext modulus is significantly larger than the plaintext modulus. The ciphertext polynomial ring (R_Q) in the BGV scheme is $C = R_Q \times R_Q$, where $R_Q = \mathbb{Z}_Q[x]/(\Phi_m(x))$, where $\Phi_m(x)$ is the mth cyclotomic polynomial with a degree of N. The relationship between m and N is determined by the Euler totient function φ, i.e., $N = \varphi(m)$. While prior works use a power-of-two N for simplicity, non-power-of-two N is

suggested for better performance and higher security flexibility. $Q \in \mathbb{Z}$ is the ciphertext coefficient modulus at level L, representing the product of several primes $(q_0, q_1, q_2, ..., q_L)$ that fit into the native integer data type. The value of Q determines the multiplicative depth, i.e., the most extended sequence of homomorphic multiplications during computation. Q is typically much larger than p, influencing the message expansion rate after encryption. The individual primes q_i are part of the modulus chain.

The BGV scheme utilizes SIMD-style processing, storing multiple integers in one ciphertext to optimize operation speed. Leveraging ring isomorphism of polynomial modulus enables multiple plaintext slots within a ciphertext. Modular arithmetic is used for homomorphic operations including addition, multiplication, and rotation. However, noise introduced during encryption limits operation numbers and requires a large ciphertext modulus (Q).

Efficient Representation of Polynomial and NTT

To handle the large ciphertext modulus Q, the BGV scheme uses a Residue Number System (RNS) format, splitting the polynomial into $L+1$ residue polynomials with coefficients under modulo q_i, where q_i's are pair-wise coprime integers. RNS allows for efficient multiplication and addition of ciphertext polynomials using current hardware systems.

To accelerate polynomial multiplication, the Number Theoretic Transform (NTT) is used, converting the polynomial to an integer Discrete Fourier Transform (DFT) representation using a twiddle factor ω that meets specific conditions. For efficient NTT and INTT, radix-2 NTT implementations are applied when N is a power of two, employing Cooley-Tukey (CT) and Gentleman-Sande (GS) algorithms. The ciphertext polynomial is represented as a matrix of polynomial coefficients in integer DFT representation of size $(L+1) \times \varphi(m)$, enabling straightforward element-wise operations for multiplication and addition between polynomials.

The BluesteinNTT algorithm is used for polynomial conversion between coefficient representation and integer DFT representation when N is a non-power of two. The algorithm requires two twiddle factors: TF1, the twiddle factors for polynomial ring m, and TF2, the twiddle factors for a power of two polynomial ring. First, the input polynomial is multiplied element-wise by TF1 to generate a polynomial C. Then, the polynomial C is padded with zero to become C_pad and then multiplied by polynomial D_pad. D_pad is a polynomial generated from TF1. Both polynomials C_pad and D_pad have length power of two greater than $2m-1$. The polynomial multiplication between them is accelerated by the radix-2 NTT algorithm (CT and GS) that requires TF2. The multiplication result (C_pad **x** D_pad) is then truncated to have length m, the exceeding coefficient added to the polynomial. The resulting polynomial is then multiplied element-wise by TF1. Finally, filter the polynomial to have a length from m to N.

Comparison Operation Algorithm

The state-of-the-art comparison algorithm for BGV/BFV was proposed in [25]. It exploits SIMD-style processing such that many comparisons can be performed in parallel, leading to a small amortized comparison latency. A large integer comparison operand is encoded into an element of $\mathbb{F}^{l}_{p^d}$. Here, $\mathbb{F}^{l}_{p^d}$ represents a finite field extension of degree d over a prime field with p elements. The encoding process involves decomposing the large integer into an element in this vector space, where the vector space is of dimension l.

To compare two integers a and b, first, each integer is *decomposed* into multiple slots in the form of $\mathbb{F}_{p^d}$. For example, a is decomposed into $a_0, a_1, \ldots, a_{l-1}$, where a_i occupying the i-th slot. For each slot, using *mod extract* step, each number is further split into multiple digits in the form of $\mathbb{F}_p$. For example, a_0 is split into a_{00} (the first digit in the first slot), a_{01} (the second digit in the first slot), etc.

To perform comparison, the algorithm first extracts digits of encrypted numbers in $\mathbb{F}_p$, then performs equality (EQ) and less than (LT) functions for each digit using specific equations. The computation of LT and EQ for each digit is independent and there is no dependency relation. The results of the equality and less than functions on the digits are combined through lexicographical order. First, the lexicographical order is computed for each block of d digits, and then the results are combined using a final equation that returns encrypted 1 when $a < b$ and 0 otherwise. The last two steps that involve ciphertext shifting and multiplying with the result from the equality circuit are called *ShiftMul*, whereas the step for performing a summation of the ciphertext is called *ShiftAdd*. The digit comparison steps are expensive due to repeated ciphertext exponentiations with large exponents for $d \times l$ times, while other steps (*Extraction*, *ShiftMul*, and *ShiftAdd*) are faster. The process represents a Bivariate circuit with separated LT and EQ computations, whereas the Univariate circuit combines LT and EQ circuits differently.

Bottleneck Analysis of Comparison Operation

The state-of-the-art BGV comparison implementation is in HElib [25]. It was reported that it was up to 3× faster than prior work based on BGV/BFV, and achieved even better performance than bit-wise FHE schemes in basic comparison tasks such as less-than, maximum, and minimum operations. However, each comparison still takes up to several seconds, hence we argue for the need for infrastructure acceleration.

To accelerate BGV comparison in HElib, we first identify the bottlenecks in the library component. To achieve this, we perform profiling and measure the execution time breakdown based on the components in the library. Figure 4.2 shows the execution time breakdown of the comparison operation based on the primitive HElib components. For brevity, the figure only shows the profiling results from the Univariate case, but we note that the Bivariate case exhibits similar results. The

platform we used for the profiling is detailed in Table 4.2.

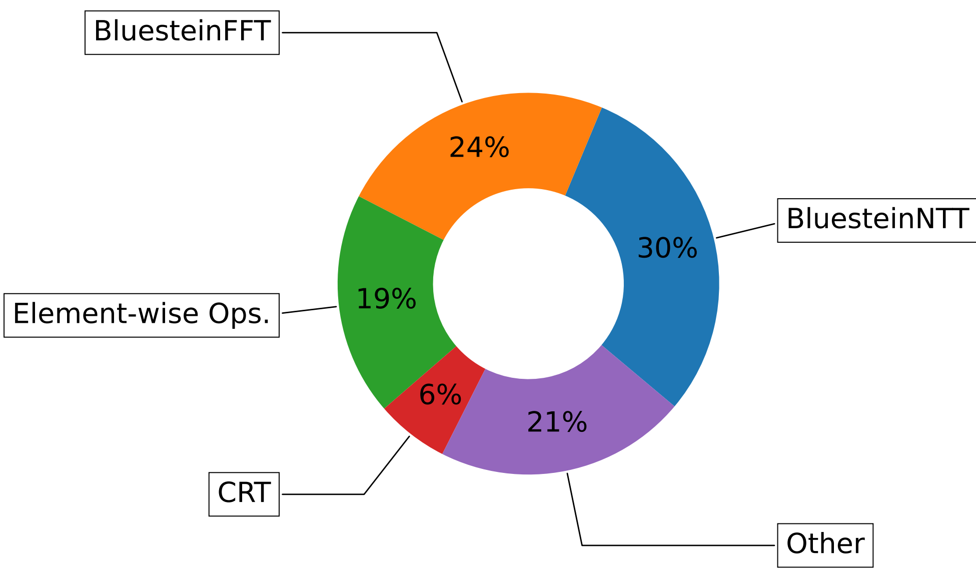

Figure 4.2: Breakdown of BGV comparison time for Bivariate circuit with parameters $m = 34511$, $p = 3$, and $d = 6$.

The figure shows that the execution time is mainly spent on three components: BluesteinNTT, BluesteinFFT, and Element-wise operations. Upon code inspection, we found that they are also highly parallelizable, so offloading them to GPU could be fruitful. In contrast, the "Other" component is also quite significant. It consists of many small loops scattered inside the library. While the code is parallelizable, the degree of the parallelism is too small to compensate for the overheads of memory copy, memory allocation, and kernel launch. Therefore, offloading these codes to GPUs may not yield net performance improvement. For CRT, although the code involves multiple loops, the most time-consuming loops in this component involve the computation of a big integer and storing the final result in it. Currently, there is no support for big integer data types on GPUs, whereas a highly optimized library for CPUs exists[21]. Therefore, both CRT and Other components may not benefit from GPU offloading; instead, we will utilize CPU for their parallelization. Note that CRT parallel execution on multiple CPU cores is already the case in HElib, hence we

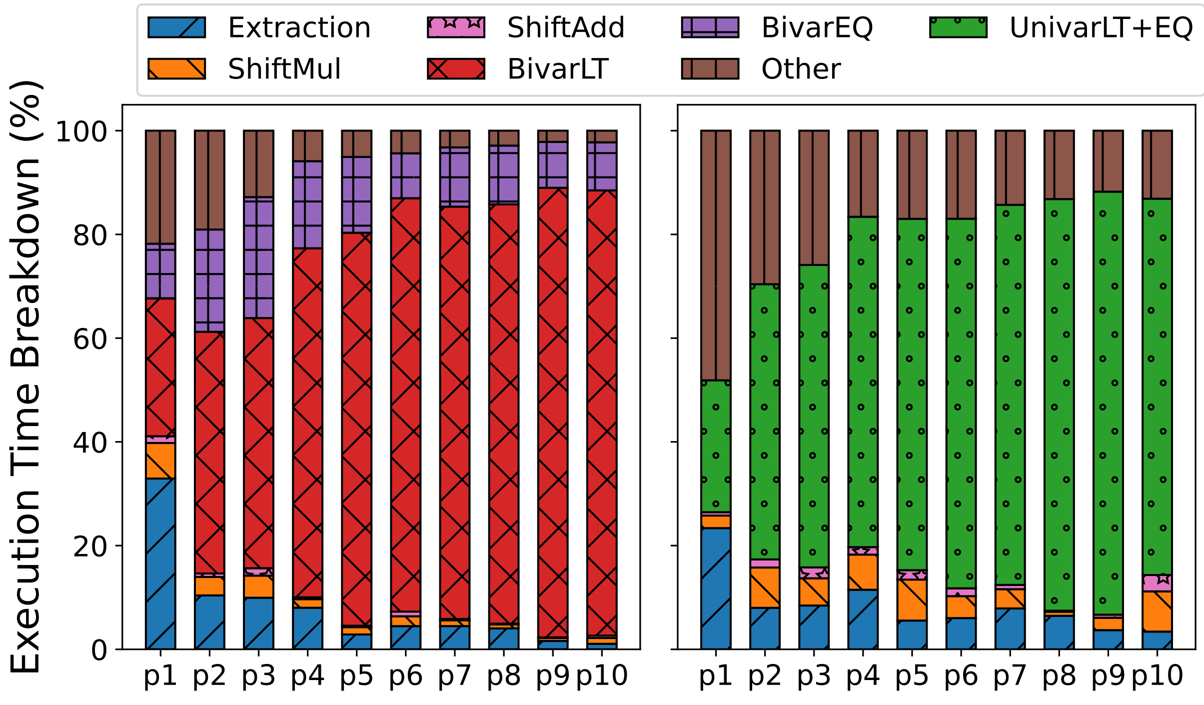

Figure 4.3: Execution time breakdown for various BGV parameters for Bivariate (left) and Univariate (right) circuits.

keep it that way. Furthermore, we add parallel execution of "Other" components on CPUs.

Next, we repeat the profiling while varying the FHE parameters, shown in Figure 4.3 with a Bivariate case on the left and a Univariate case on the right. We tried ten different sets of parameters (p1 to p10), with p increasing from 3 to 31 (details in Table 4.2). Being the plaintext modulus, the parameter p determines the size of the plaintext space, i.e. larger p increases the maximum number that can be encoded in the plaintext space. For the Bivariate Circuit, *BivarEQ* and *BivarLT* are the most time-consuming steps in the algorithm. Moreover, with the increase of the size of parameter p, the dominance of these steps increases. For the Univariate Circuit, the *UnivarLT+EQ* steps dominate most of the computation time and increase as p increases. The steps are computationally

expensive due to repetitive ciphertext exponentiations performed in a loop. Each exponentiation has parallelism that could benefit from running on a GPU, and indeed HElib offloads it to the GPU. However, they are still very expensive, and thus further optimizations are needed.

The Design of BoostCom

This section describes BoostCom, our solution for BGV comparison operation acceleration through the use of GPU and multiple CPU threads. After we conduct the execution time breakdown from the previous section, we discover some primitive components inside the HElib that need to be offloaded to the GPU and what steps in the algorithm to look out for the possibility of acceleration with multi CPU threads.

Comparison-Friendly BGV Parameter Tuning

The conventional wisdom for BGV parameter selection is that the degree of polynomial ring should be a *power-of-two* (PoT) number, as reflected in most prior works. However, it is noteworthy that BGV can also accommodate a polynomial ring with a non-PoT degree. When the polynomial ring possesses a non-PoT degree as a result of choosing a prime number or a product of a few prime numbers of the cyclotomic polynomial ring order, it ensures that the slot permutation group is cyclic or a composite of several cyclic groups, leading to enhanced performance [25, 19].

The choice of parameters affects the end-to-end performance, security, and computation efficiency trade-offs and degree of SIMD parallelism. However, ignored by the conventional wisdom is the implication of selecting a PoT polynomial ring degree on the performance of BGV comparison. For example, while homomorphic operations are slightly faster with PoT polynomial rings, comparison is multiple orders of magnitude slower with PoT. Since prior works overlooked comparison, the

highly adverse impact of PoT ring degree did not surface.

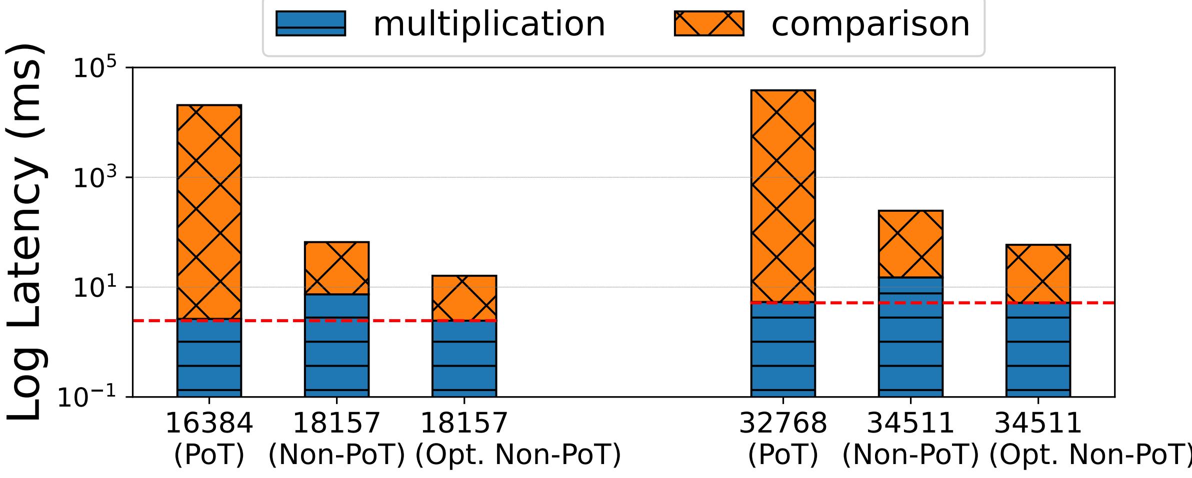

Figure 4.4: Comparison of the homomorphic operation latencies of using the power of two vs. non-power of two vs. optimized non-power of two polynomial rings.

As a demonstration, Figure 4.4 presents stacked bars representing the latencies of a single multiplication and a single comparison in *logarithmic scale* for a pair of m values. To qualify this, while ensuring a security level of $\lambda > 128$ bits, we have chosen two non-PoT m values, 18,157 and 34,511, to correspond with two specific PoT m values, 16,384 and 32,768. To ensure a meaningful comparison, these selections have been designed such that the non-PoT m values yield a slightly larger count of SIMD slots and an enhanced security level, as guided by the recommendations from [4, 25]. For each m value, we show the stacked latencies for three cases: PoT, unoptimized non-PoT, and non-PoT with our optimizations that will be described later. The figure shows that the stacked non-PoT latencies are more than two orders of magnitude faster than PoT, because comparison is 351.6× and 165.9× slower for m values of 16,384 and 32,768 respectively compared to the corresponding non-PoT.

It is important to note that while unoptimized non-PoT achieved much faster comparison in ex-

change of slightly slower multiplication, after optimizations that we propose, the multiplication becomes faster than with PoT, leading to non-PoT as strictly faster for each operation compared to PoT. The reason for this outperformance of non-PoT is that our optimizations accelerate the building blocks of comparison that also serve as the building blocks for addition and multiplication.

In addition to the outperformance, choosing a non-PoT polynomial ring improves flexibility and security. For instance, increasing m (cyclotomic ring order) while keeping Q (prime moduli product) constant results in a higher security level but slower computation. Thus, the choice of m presents a trade-off between security and computation time. Restricting m to PoT leaves us with limited polynomials to choose from, such as $x^{256}+1, x^{512}+1, x^{1024}+1$, etc. This may in turn force the ciphertext to be too large, lowering efficiency. If instead, we allow non-PoT polynomial rings, there are many more polynomials to choose from to rightsize the ciphertext, while still meeting a minimum security level.

Finally, the size of SIMD slot (n_{slot}) in BGV is determined by $n_{slot} = \frac{\varphi(m)}{d}$, where d is in the order of $p \mod m$. Thus, increasing n_{slot} requires decreasing d. When m is restricted to PoT, we can only vary p to affect the ciphertext size. In contrast, when m is non-PoT, it can be varied to increase the SIMD slots, allowing a higher degree of parallelism which yields a smaller amortized execution time.

Overall, choosing a non-PoT polynomial ring order is advantageous, as it leads to a much faster comparison, higher computation efficiency, and a higher degree of SIMD parallelism.

Hybrid CPU/GPU Parallelization

Profiling results (Section 4) identified BluesteinNTT, BluesteinFFT, and element-wise operations in BivarLT/BivarEQ/UnivarLT+EQ as taking roughly three quarters of the execution time. Thus,

an obvious acceleration step is to offload them to the GPU to benefit from the massive parallelism on the GPU. However, after offloading, through profiling we found that the GPU utilization is less than 10%. Meanwhile, the CPU is mostly idle waiting for GPU computation results. To address both problems, we propose hybrid parallelization where higher-level parallelization is performed at the CPU.

To perform parallelization on the CPU, one approach is to only parallelize the most time-consuming operations (i.e., ciphertext exponentiation). However, this approach is challenging as the use of recursion creates loop-carried dependences. Moreover, the exponent of the parameters depends on p, which may exceed the number of CPU threads, making load balancing challenging. Hence, we explore an alternative approach of parallelizing across digits. As discussed earlier in 4, the computation of each digit in LT and EQ has no dependence on the computation of other digits. LT_{ij} computes LT with digit input a_{ij} and b_{ij} only. The computation of each digit is also highly parallel. Hence we adopt a hybrid parallelism strategy, where we use GPU for specific computations for each digit in parallel, and utilize multicore CPUs to exploit digit-level parallelism. This is illustrated in Figure 4.5.

The parallelization for digit computation is wrapped inside a library which we name cuHElib, built on top of HElib. We added multiple buffers (called CuBuffers) to hold data in the GPU memory, a command queue to dispatch tasks to the GPU, and changed the GPU task offloading strategy. The library offloads each expensive operation or function as a task (BluesteinNTT, BluesteinFFT, and Element-wise operations) to the GPUs. At the digit level, the computation of LT and EQ of different digits are computed across multiple CPU threads simultaneously. To avoid races and synchronization, we allocate a separate GPU buffer for each CPU thread.

Since the CPU and GPU have separate memories, offloading computation tasks to the GPU requires copying data to a GPU buffer, launching a kernel to compute the task, and then copying the result

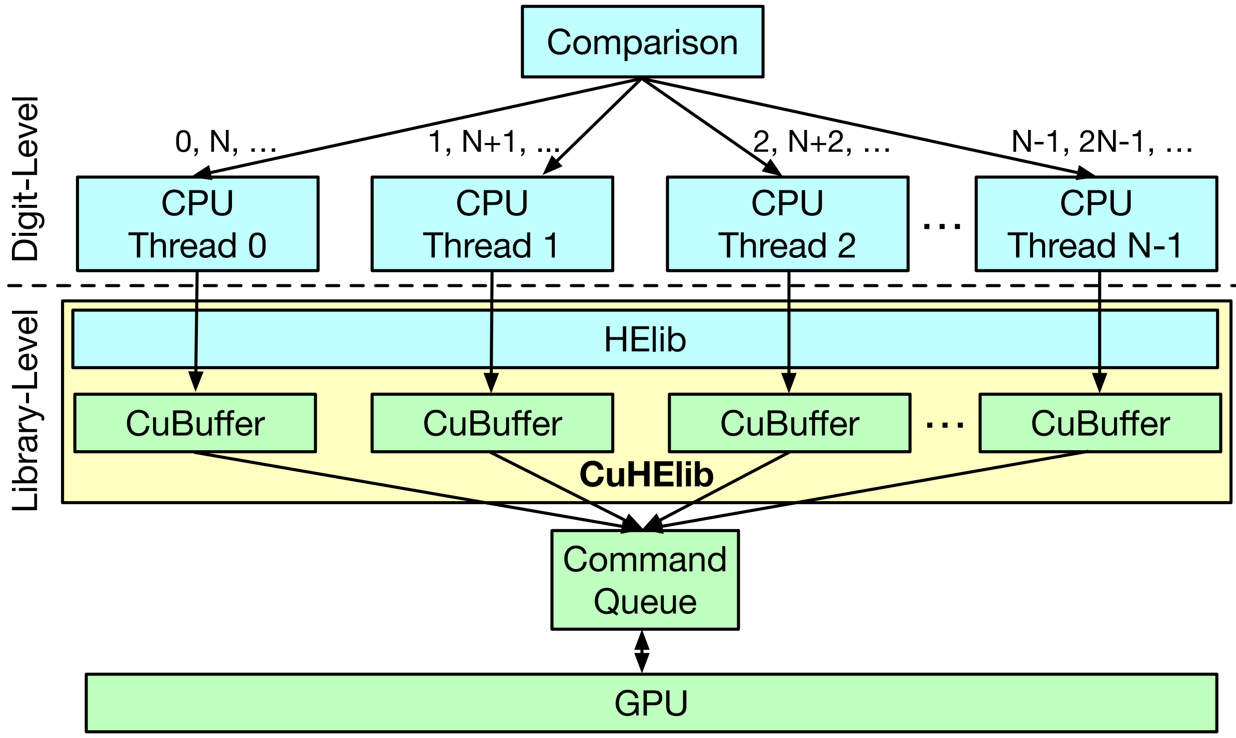

Figure 4.5: Illustrating Boostcom's hybrid parallelism: digits are computed across multiple CPU threads, while primitive operations in each digit are offloaded to the GPU.

back to the CPU. If Unified Memory (UM) [48] is supported, the copying may be performed implicitly as the CPU and GPU share virtual memory address space. However, to avoid page thrashing and page faults while a kernel is running, we use explicit copying with careful timing.

Slot Compaction

SIMD-style processing facilitates the simultaneous manipulation of tens of thousands of numbers encoded within a single ciphertext, whereby an operation on the ciphertext is performed on all encoded numbers. Thus, a high slot utilization increases both compute and memory efficiency. However, our analysis reveals that slot utilization is often low, especially for comparison, due to

two reasons. First, there are often discrepancies between the input size alignment and the available ciphertext slots, which persist even after optimizations. For instance, AlexNet's input size is 224×224, while SEAL [44] supports a maximum of 16,384 slots per ciphertext. Consequently, the input is partitioned into $\lceil \frac{224 \times 224}{16384} \rceil = 4$ ciphertexts, resulting in 76% slot utilization. The presence of unused slots presents an opportunity for compaction.

A second reason for slot under-utilization is less straightforward; it is the result of performance optimizations (standard in Helayers[2]) for non-comparison operations where some numbers are duplicated in different slots. Figure 4.6 illustrates an example of a neural network where a convolution operation (and batch normalization) precedes ReLU in which comparison is performed (part (a)). To perform the convolution between a matrix M and a filter (part (b)), the matrix M fills up all 16 ciphertext slots, while the convolution filter is duplicated four times to fill up slots (part (c)). The purpose of filter duplication is to reduce the number of multiplications and rotations. In order to obtain the convolution results, a multiplication is followed by three sets of rotate-and-accumulate. The resulting result ciphertext has the convolution results in the 4th, 8th, 12th, and 16th slots (shown in grey), while all other slots do not contain useful values.

Next, Figure 4.6 (part (d)) illustrates the state-of-the-art practice where each convolution result number is split into its digits (four digits per number are illustrated). Thus, each useful slot in the convolution result becomes four new slots across four different chiphertexts, each new slot representing a digit. Note, however, that useless numbers are also decomposed into digits, resulting in 75% of the slots not containing useful data. When performing comparison, only the slots with useful digits are needed, which presents an opportunity for compaction. Our approach is shown in part (e), where we consolidate digits from all numbers into a single ciphertext. Through slot compaction, the comparison can now work on fewer ciphertext inputs, substantially reducing memory usage and unnecessary computation.

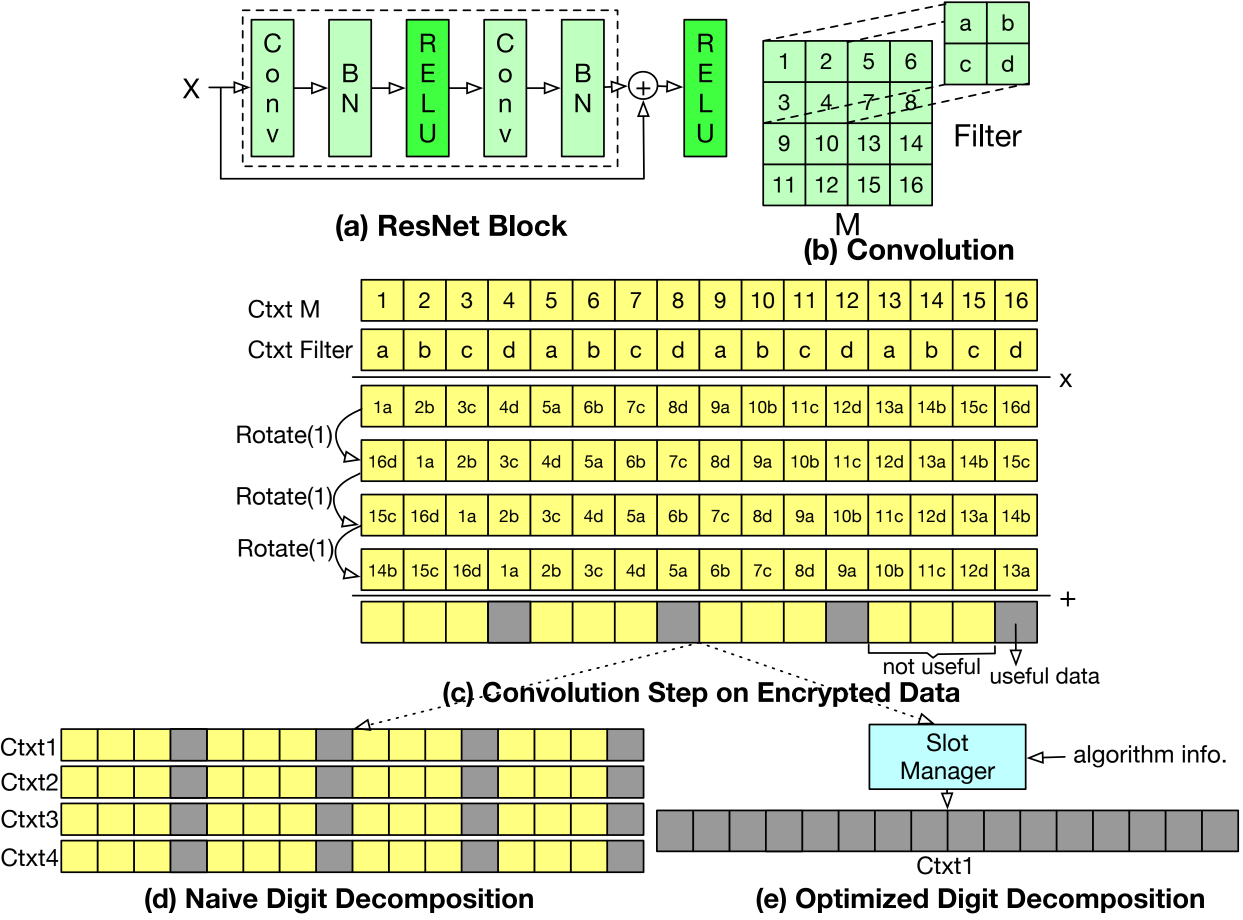

Figure 4.6: Illustration of: (a) ResNet block containing convolution (Conv), batch normalization (BN), and ReLU; (b) Convolution filter; (c) Convolution steps on encrypted data resulting in unused slots; (d) Naive digit decomposition with many unused slots; and (e) Optimized digit decomposition with slot compaction.

Achieving slot compaction in the Helayer is difficult because the information of which slots contain useful data is not available to the Helayer, hence it must conservatively assume that all slots are useful. Besides, the existence of non-useful slots arises only in the case where comparison is preceded by certain operations like convolution or matrix multiplication, so the Helayer cannot predict slot usefulness without algorithmic information. To overcome this challenge, our slot manager (SM) preserves algorithm information to track slot usefulness to guide slot compaction after

digit decomposition. SM minimizes memory usage by distributing digit decomposition across as few ciphertexts as possible.

Note that the slot compaction opportunity requires three conditions: (1) the existence of comparison, (2) the comparison being preceded by other operations such as convolution or matrix multiplication, and (3) slot usefulness information being tracked and passed to the digit decomposer. The presence of the three conditions enables the SM to significantly reduce the number of ciphertexts for comparison input. However, when at least one of the conditions is absent, we just perform slot compaction for the case when the input size does not align with the ciphertext format.

Non-Blocking Comparison

When many numbers are compared together, the cost of comparison operation could be amortized using BGV SIMD-style processing. However, when an application only needs to compare a pair of numbers (or a small number of pairs), comparison latency is hard to amortize. This case occurs when the comparison occurs inside an *if* statement.

Listing 4.1 shows an example code that performs a query without FHE (i.e., on unencrypted data). It takes three inputs: query type (`q`) and two data operands (`op1` and `op2`). The code performs an operation (addition, multiplication, or exponentiation) based on the query type, with operand value specified by one of the two data operands. It has three comparisons each involving a pair of numbers. The semantic-equivalent FHE version is shown in Listing 4.2. With FHE, the query type is not in plaintext form, hence we must use the EQ(.) function to test for equality. Furthermore, the comparison results are also in ciphertext, hence conditional branches are replaced by code straightlining, resulting in Listing 4.2.

To hide the comparison latency that is hard to amortize, we propose *non-blocking comparison*.

When the comparison is solely used to determine the taken branch path, there is no dependency relation between the main computation and the branch evaluation. Consequently, we can execute the branch evaluation and the main computation concurrently. Listing 4.3 shows the resulting code with our *non-blocking comparison* optimization. The branch evaluation that computes equality functions EQ(.) is performed by a helper thread in parallel to the arithmetic operations performed by the main thread. The final data update is performed after the helper thread joins the main thread.

Listing 4.1: Private query on plaintext data.

```
privateQuery(q, op1, op2){
 if(q == add)
    Data += op1
 else if(q == mult)
    Data *= op1
 else if(q == power)
    Data = Data^op2
 else
    Data = Data }
```

Listing 4.2: Private query's straight-lined code on encrypted data.

```
privateQuery(q, op1, op2){
 c1 = EQ(q, add)
 c2 = EQ(q, mult)
 c3 = EQ(q, pwr)

 Data1 = Data + op1
 Data2 = Data * op1
 Data3 = Data.Power(op2)
 Data = Data1 * c1 + Data2 *
        c2 + Data3 * c3 }
```

Listing 4.3: Private query with non-blocking comparison.

```
EvalBranch(c1, c2, c3, q){
 c1 = EQ(q, add)
 c2 = EQ(q, mult)
 c3 = EQ(q, pwr)
}
privateQuery(q, op1, op2) {
 helper_thread(EvalBranch(c1, c2, c3, q))
 Data1 = Data + op1
 Data2 = Data * op1
 Data3 = Data.Power(op2)
 thread_1.join()
 Data = Data1 * c1 + Data2 * c2 + Data3 * c3 }
```

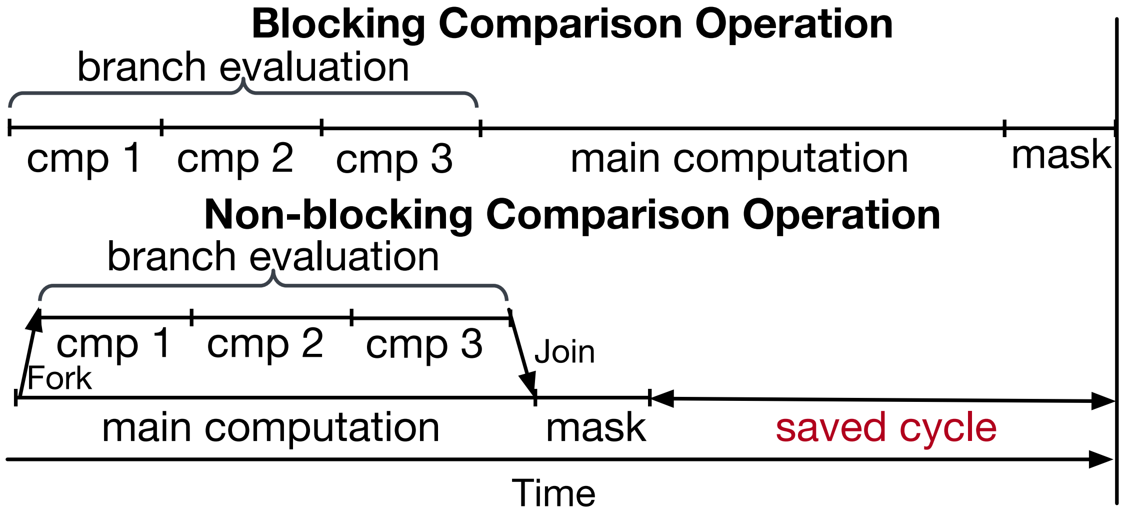

Figure 4.7: Illustrating the saved cycles due to the non-blocking comparison optimization.

To illustrate the benefit, Figure 4.7 compares the original straightlined code performance (top) vs. with our non-blocking optimization (bottom). With non-blocking, the execution of branch evaluation overlaps with the main computation.

BluesteinNTT Acceleration

BluesteinNTT is a time-consuming function in HElib, transforming a ciphertext polynomial from a coefficient representation into an integer DFT representation. The BluesteinNTT computation involves element-wise multiplication (2 times), radix-2 NTT/INTT conversion, element-wise addition, and polynomial filtering. To accelerate it, we adopted the state-of-the-art radix-2 NTT/INTT implementation [47], applied an optimization [64], and used the Barret reduction for modular operations [54]. We discovered that the remaining performance bottleneck is in polynomial filtering, which is not parallelizable due to loop-carried dependency.

Polynomial filtering alters the polynomial length from m to N. In Listing 4.4, the update of the vari-

able j is control-dependent on the loop iterator i, creating a loop-carried dependence that hinders loop-level parallelization. If executed sequentially with a single GPU thread, it would be inefficient due to the comparatively slower speed of a single GPU thread compared to a CPU thread. Instead, we propose a *branch removal* optimization by breaking down the code into two phases (Listing 4.5): the *offline phase* and the *online phase*. The offline phase removes loop-carried dependences by pre-computing indices to set the target index for *final_result*. This is achieved by computing the prefix-sum of the value array of *ZmStar*. Additionally, since all the inputs for index pre-computation are available before FHE computation, we can pre-compute it on the CPU. As a result, the online phase, when it performs selective copy, becomes parallelizable as we remove the branch and can benefit from GPU execution. This transformation also leverages efficient GPU pipeline computation and enables the use of multi-streaming to further improve GPU utilization.

Listing 4.4: BluesteinNTT polynomial filtering code showing loop-carried dependence due the if statement and j++.

```
1 for (i = 0, j = 0; i < m; i++)
2   if (zMStar->inZmStar(i))
3     final_result[j++] = coeff(result, i);
```

Listing 4.5: *Branch removal* optimization that removes loop-carried dependence in polynomial filtering.

```
1 //offline phase: index pre-computation to remove loop-carried dependence
2 prefixSum(sumZmStar, inZmStar, getM);
3 //online phase:selective copy executed in parallel with GPU
4 __global__ filterBluestein(tmp, inZmStar, sumZmStar, m){
5     int i = blockDim.x * blockIdx.x + threadIdx.x;
6     if (i < m && inZmStar[i] != 0)
7       final_result[sumZmStar[i]] = result[i]; }
```

BluesteinFFT Acceleration

BluesteinFFT significantly contributes to the comparison operation latency in HElib. To ensure the correctness of the ciphertext, HElib checks the noise level after each operation using BluesteinFFT. While one could use a very large Q value to prevent noise budget exceedance, this may reduce computation efficiency. Opting for smaller Q values, though requiring noise estimation using BluesteinFFT, may enhance computation efficiency.

HElib utilizes the CPU library PGFFT, which we replace with the cuFFT library for GPU offloading. Before using BluesteinFFT with cuFFT, a configuration step is necessary, involving plan creation for optimal thread organization. Two distinct strategies are under consideration to optimize the utilization of cuFFT: the first involves the creation of the execution plan before every BluesteinFFT operation, a straightforward yet computationally expensive approach; the second strategy, denoted as *plan reuse*, configures the plan once at the initiation of FHE computation. Subsequently, during the execution of BluesteinFFT, pointers for twiddle factors and the GPU execution plan are conveyed, effectively eliminating the need for plan creation on the critical path of the operation.

Accelerating Element-Wise Operations

Element-wise operation in HElib multiplies two matrices of size $(L+1) \times \varphi(m)$ by iteratively multiplying and adding. Each matrix is dynamically allocated because a homomorphic operation may add and/or delete rows during execution. The dynamic allocation may result in non-contiguous memory addresses, which creates a problem for *cudamemcpy* which only copies contiguous memory address range. Thus, copying an entire matrix to the GPU using *cudamemcpy* may lead to copying unrelated data. Moreover, copying the matrix result back is not feasible, as it may over-

write unrelated data processed by other threads. We explore several options to address the issue.

One possible approach is to perform the element-wise operation row-by-row, similar to Intel HEXL library [7]. However, Intel HEXL relies on CPU. If we use this approach for GPU, we may suffer from high memory copying latencies for each row of the matrices and from a low degree of parallelism on the GPU, which may result in underutilized GPU. An alternative approach involves copying the entire matrix to the GPU row-by-row and then executing the element-wise operation for the entire matrix. This generally reduces the total kernel time. However, it still results in PCIe bandwidth wastage since only a small amount of data is copied to the GPU multiple times.

Thus, we use a third approach, which we refer to as *layout transformation*, to create a contiguous memory allocation in the CPU buffer, which allows the element-wise operation for the entire matrix to be offloaded in a single GPU kernel. Figure 4.8 illustrates the steps for this optimization: ① data from the original matrix with non-contiguous row locations is copied over to a new buffer with contiguous mapping. ② the entire matrix with contiguous rows is transferred to the GPU. ③ element-wise operation is performed on the GPU, producing results in the GPU buffer. ④ data in the GPU buffer is copied back. ⑤ the resulting matrix is copied back to the original buffer. This approach incurs additional CPU-to-CPU memory copying but maximizes PCIe bandwidth utilization and allows a high degree of GPU parallelization.

Methodology

Experiment Platforms

We evaluate BoostCom on a combination of GPU and CPU platforms. The GPU platform has an NVIDIA RTX 3090 GPU with 82 Streaming Multiprocessors (SMs). Each SM contains 128 CUDA cores, operating at a core clock speed of 1695 MHz. The GPU is equipped with a combined

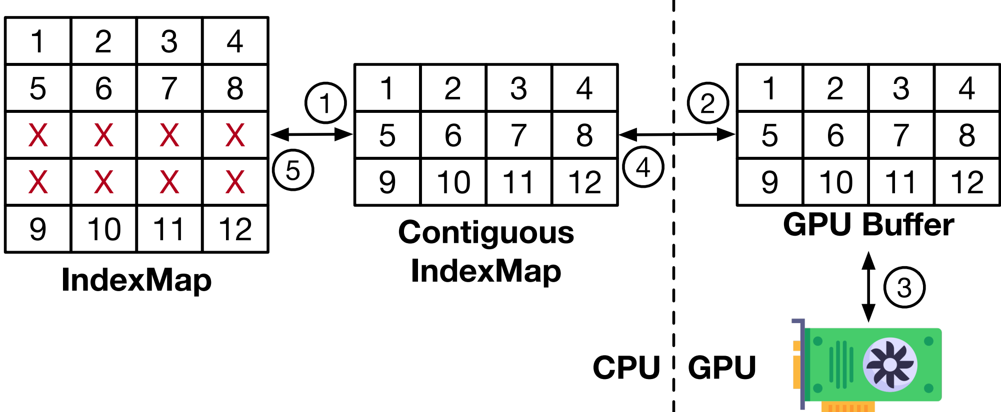

Figure 4.8: Layout optimization for offloading the element-wise operation to the GPU, utilizing additional copying at the CPU side to maximize the CPU-GPU *memcpy* bandwidth and parallelization degree.

10 MB of L1 data cache and shared memory, along with a separate 6 MB L2 cache. The GPU memory system has a 24GB size and 936 GB/s bandwidth.

The CPU platform has an AMD Ryzen PRO 3955WX CPU with 16 cores and 128 GB of memory. Each core has a clock speed of 3.9 GHz, with a 4.3 GHz maximum turbo frequency. It has 64MB L3 Cache Memory and its main memory has eight-channel ECC DDR4-3200 DRAMs. The CPU runs Ubuntu OS version 22.04 and NVIDIA driver version 525.85.12. We used CUDA version 12.0 and GCC version 7.5.0 for compilation.

Workload Evaluation Methodology

We evaluate BoostCom with full applications to measure overall application performance as well as with microbenchmark to measure comparison performance specifically. In both cases, our evaluation measures end-to-end performance, in contrast to extrapolating from the measurement of each operation that is common in prior works. End-to-end performance measurement gives a

fuller and more reliable picture of the performance. For all measurements, we repeat each experiment 10 times and report their average. We use the NVIDIA Nsight system to collect hardware performance statistics.

Microbenchmark

To measure comparison-only performance, we form a microbenchmark that performs a comparison of a pair of 64-bit integers. We vary the BGV parameters to form 10 different configurations following prior work [25]. Each configuration is expressed as a tuple of $(p\ m\ N)$ and was selected to maximize the number of SIMD slots as shown in Table 4.2. They are sorted in the order of increasing plaintext modulus p values. Each configuration uses bivariate and univariate circuits with differing vector space dimension d, vector length l, and the product of prime moduli Q. The resulting security level λ and number of integers that can fit in one ciphertext are shown in the last two columns.

Table 4.2: Parameters and Statistics

Params	$(p\ m\ N)$	Circuit	$(d\ l)$	log(Q)	λ	no of int
p1	(3 34511 34510)	B	(6 7)	324	298	290
		U	(16 4)	472	189	507
p2	(5 19531 19530)	B	(7 4)	324	155	697
		U	(7 6)	354	141	465
p3	(7 20197 19116)	B	(6 4)	354	137	531
		U	(8 4)	406	110	531
p4	(11 15797 15796)	B	(5 4)	342	162	359
		U	(5 5)	378	145	287
p5	(13 30941 30940)	B	(5 4)	354	338	1547
		U	(4 6)	378	313	1031
p6	(17 41761 41760)	B	(4 4)	413	402	1305
		U	(7 3)	472	344	1740
p7	(19 29989 29988)	B	(4 4)	378	302	833
		U	(5 4)	385	296	833
p8	(23 37745 30192)	B	(5 3)	413	275	838
		U	(9 2)	456	245	1258
p9	(29 18157 17820)	B	(5 3)	360	175	990
		U	(6 3)	413	150	990
p10	(31 52053 34700)	B	(5 3)	512	252	2313
		U	(4 4)	512	252	1735

Applications

We used the following applications that utilize BoostCom for all BGV operations using our cuHElib including comparison.

sorting is an application that sorts an array of 16 encrypted 32-bit integers from [25]. It uses univariate circuit for comparison, utilizes a matrix of Hamming weights to establish the relationship between any pair of elements in the encrypted array.

min is an application that finds a minimum integer from an array of 16 elements of 32-bit integers from [25]. It uses univariate circuit and combines the Hamming weight matrix and the tournament methodology, reducing the circuit's depth for improved efficiency.

mlp is a simple machine learning program utilizing Multi-Layer Perceptron that we wrote to classify images. It has three layers: a fully-connected layer, ReLU, and another fully-connected layer. In contrast to prior works that usually did not support ReLU, by supporting comparison, we allow ReLU operation. The bivariate circuit is used for comparison in the ReLU layer. mlp performs inference using encrypted 16-bit integers. The input image has 28x28 pixels, stored in a single ciphertext. It trains on MNIST datasets and outputs ten nodes.

img_col is an image recolorizing application that we developed to calculate the distance of every pixel inside an image to a threshold value. When the distance is below the threshold, it transforms the pixel by multiplying its color value with the pre-set value. The bivariate circuit is used for comparison. This application enables private medical data image analysis on an untrusted cloud server. The input is an encrypted image, threshold value, and pre-set pixel transformation value. The input image is encoded into 16 ciphertexts and each ciphertext consists of 700 pixels.

private_q is a simple application that we developed to perform a private query to manipulate data

Table 4.3: Memory Usage for each Workload (GB).

	sorting	min	mlp	img_col	private_q	gmean
BoostCom	5.5	8.4	1.6	2.8	2.2	3.5
BoostCom+SM	4.5	6.5	1.1	1.6	1.4	2.5
Mem. Reduction	19%	23%	32%	44%	35%	29.3%

in encrypted databases, based on Listing 4.7. This application helps evaluate the proposed *non-blocking comparison*.

Evaluation Results

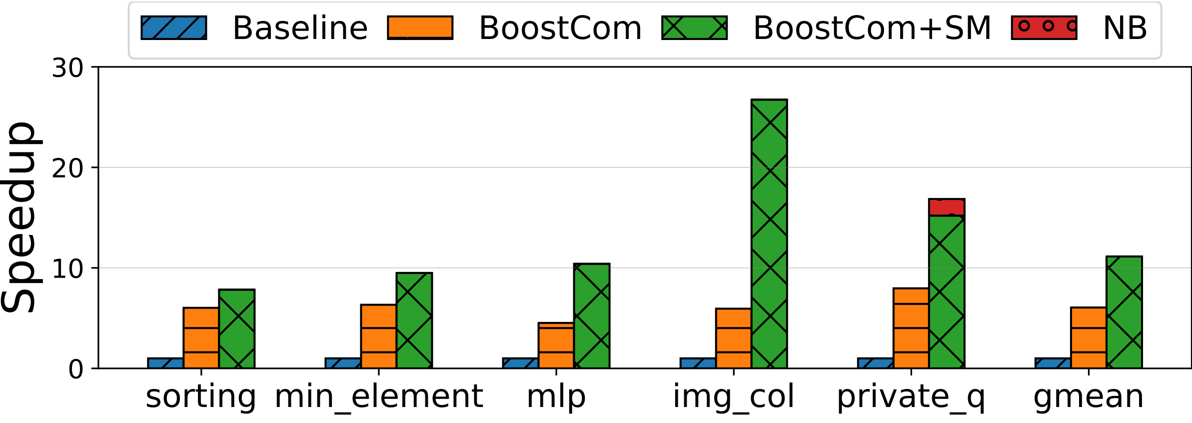

Figure 4.9: The acceleration achieved by BoostCom in comparison to the baseline for five important workloads.

Workloads Speedup

Figure 4.9 illustrates the speedups achieved by various levels of BoostCom optimizations compared to a 16-core CPU-only baseline (i.e., HElib [22]), for all applications and their geometric mean

speedup. Basic BoostCom (second bars) include library-level optimizations, i.e., hybrid CPU/GPU parallelization, BoostCom+SM (third bars) adds a slot manager, and NB (fourth bars) adds the non-blocking optimization.

On average, the basic BoostCom achieved a gmean speedup of 6.0× over the baseline. It is important to note that the speedups are measured for *end-to-end execution times*, encompassing various operations, not just the comparison. This includes all overhead such as CPU-GPU memory copy, kernel launches, synchronization, etc. The gmean speedup nearly doubles when slot compaction is added, reaching 11.1×. This demonstrates the effectiveness of slot compaction that reduces the number of ciphertexts involved in comparison. Finally, the non-blocking optimization adds some speedup to private_q.

The effectiveness of slot compaction is due to both memory usage and computation reduction from working with fewer ciphertexts. Table 4.3 shows the reduction in memory usage (29.3% on average), which clearly correlates with the speedups; the greater the memory usage reduction, the higher the speedup.

In the subsequent subsection, we analyze the effect of each optimization at the library level employed in BoostCom concerning only the comparison operation.

Comparison Operation Speedup

Figure 4.10 compares the end-to-end execution time of comparison of encrypted 64-bit integers, over the 16-core CPU-only baseline for Bivariate (top) and Univariate (bottom) circuits, across 10 different BGV configurations from Table 4.2. For each configuration, six bars are shown with increasing optimization levels, starting from the baseline, layout transformation, branch removal, plan reuse, the combination of three said optimizations (*all*), digit level parallelization with CPU

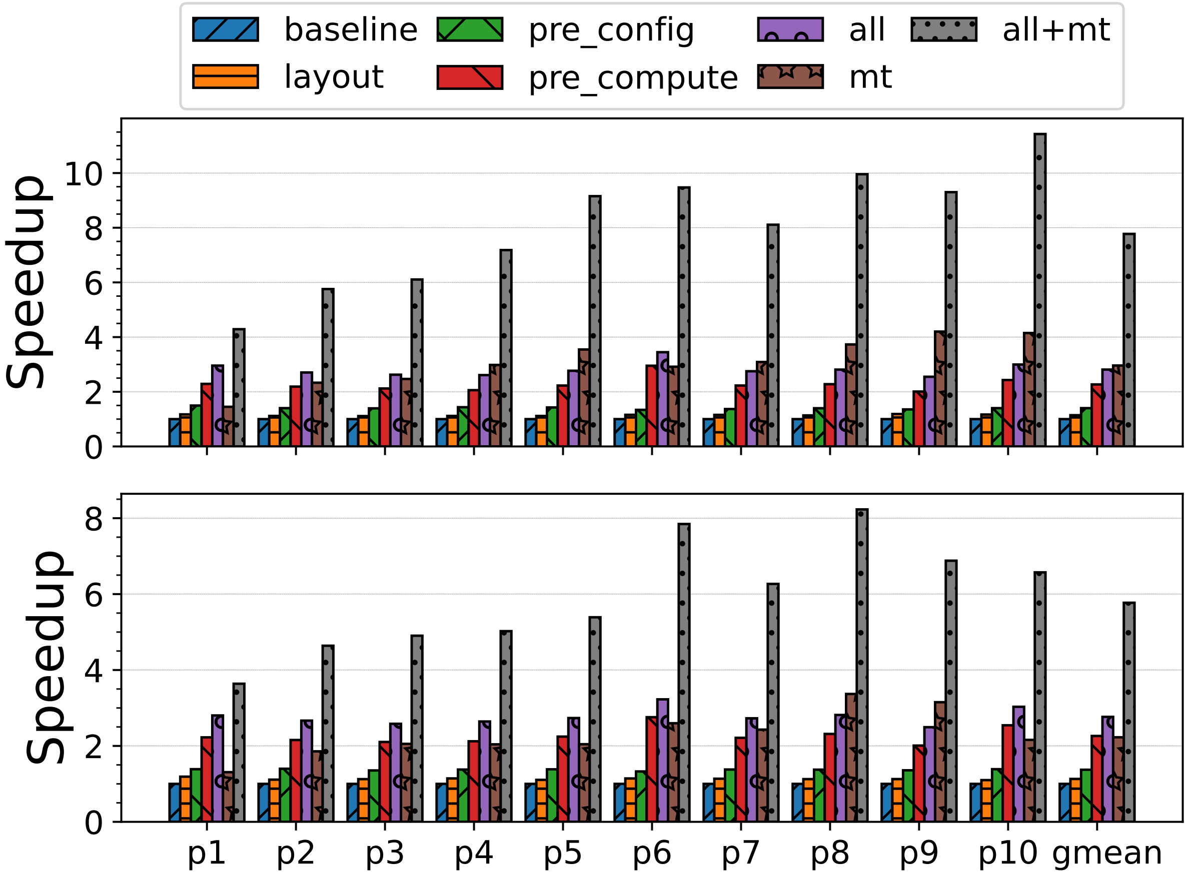

Figure 4.10: Speedups of the comparison operation for the Bivariate (top) and Univariate (bottom) circuit over the 16-core CPU baseline.

multithreading (*mt*), and all optimizations including multithreading (*all+mt*). Note that slot compaction and non-blocking comparison optimizations are not applicable here since there is no other computation aside from the comparison itself.

For both circuits across all configurations, each optimization adds additional speedups, indicating their effectiveness. With *all*, the geometric mean (gmean) speedup is 2.8× for both circuits. On its own, multithreading for digit-level parallelization is somewhat effective (gmean speedup of 2.9×

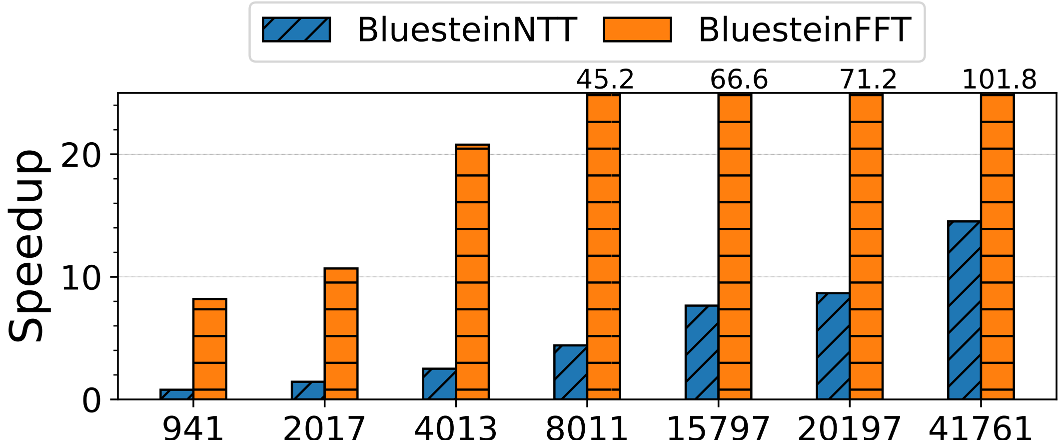

Figure 4.11: The comparison between BluesteinNTT and BluesteinFFT speedup over each baseline with the increasing parameter m.

(Bivariate) and 2.2× (Univariate)). But when combined with all other optimizations, multithreading enables much higher speedups, reaching 7.8× (Bivariate) and 5.8× (Univariate), due to the synergistic effect where multithreading significantly improving the GPU utilization (by between 30% and 260%).

Roughly, as p increases, the effectiveness of multithreading increases whereas that of other optimizations remain unchanged. This is because as the degree d increases, the fraction of execution time spent on the *BivarCircuit*, *EqualityCircuit*, and *UnivarCircuit* increases (as shown in Figure 4.3).

BluesteinNTT and BluesteinFFT Sensitivity Study

Increasing multiplicative depth without sacrificing security may lead to larger m. To evaluate its effect on BoostCom, we vary m from 941 to 41,761, resulting in polynomial size expansions ranging from 2,048 to 131,072. The resulting speedups of BluesteinNTT and BlusteinFFT, calculated

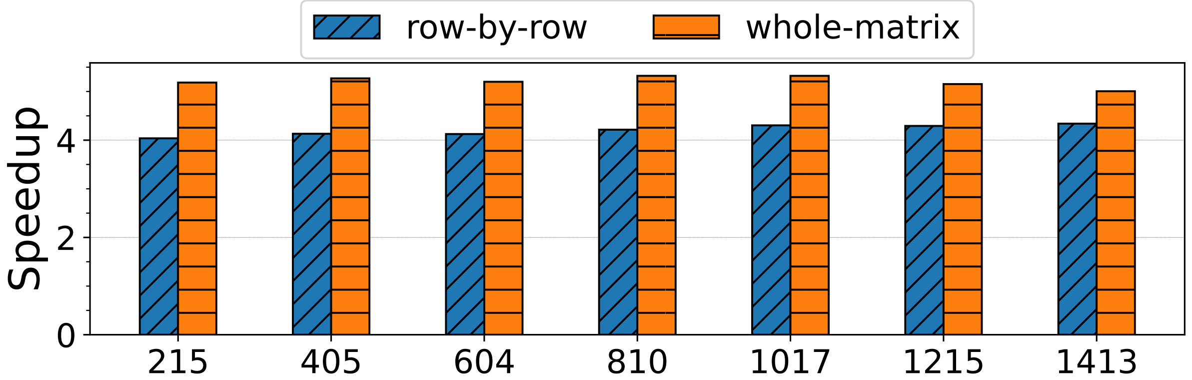

Figure 4.12: Element-wise operation speedups of BoostCom's whole matrix approach vs. row-by-row GPU offloading, as Q (number of bits) increases.

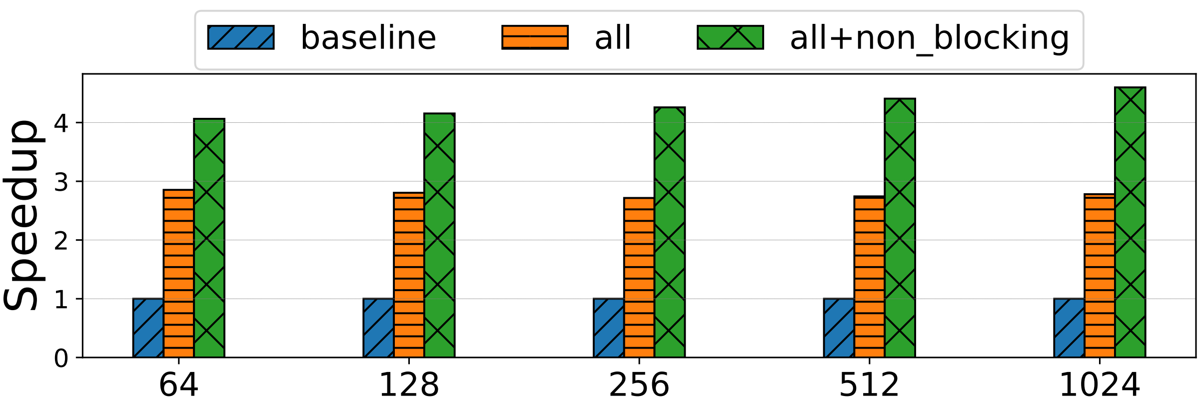

Figure 4.13: Speedups of optimizations without vs. with non-blocking as the branch evaluation computation increases with larger exponent values.

over CPU-only execution are shown in Figure 4.11 (top). The figure shows that the larger the m, the higher the speedups, indicating BoostCom's scalability.

Element-wise Sensitivity Study

Figure 4.12 compares the speedups of BoostCom's *layout transformation* compared to performing element-wise operation row-by-row, as Q increases. A larger Q increases the noise budget and allows a more complex application but with slower computation. The figure shows that the speedups of our layout optimization is quite stable across all values of Q.

Non-blocking Comparison Sensitivity Study

To evaluate the sensitivity of BoostCom's *non-blocking* optimization performance, we vary the exponent (`op2`) from 64 to 1024 as exponentiation is the most expensive operation. (Figure 4.13). The figure shows the speedups are stable, with increasing non-blocking effectiveness (as a larger portion of the branch evaluation is hidden).

Conclusion

In this study, we proposed accelerating fully homomorphic BGV scheme on CPU/GPU systems, via novel optimizations that include comparison-friendly parameter tuning, hybrid CPU/GPU parallelization, slot compaction, non-blocking comparison semantic, as well as other optimizations applied to BluesteinNTT, BluesteinFFT, and element-wise operations. The amalgamation of the proposed optimizations demonstrated significant effectiveness, yielding a speedup of 11.1$\times$ (up to 26.7$\times$) across five crucial FHE applications when compared to an industry-level FHE library running on a 16-core CPU.

CHAPTER 5: BOOSTING FHE COMPUTATION VIA CIPHERTEXT COMPRESSION

In this chapter, we address the challenges of accelerating privacy-preserving computation through ciphertext compression.

Introduction

Fully Homomorphic Encryption (FHE) represents a groundbreaking advancement in secure computing by enabling confidential computations without the necessity of sharing secret keys. This innovation relies on the manipulation of high-degree polynomials with substantial coefficients. To support intricate applications and bootstrapping, these coefficients typically range from 1600 to 2000 bits in size. In FHE, data undergoes encoding into plaintext polynomials before encryption into ciphertext, comprising pairs of large polynomials, upon which FHE computations directly operate.

However, despite its promise, the current FHE scheme encounters challenges, particularly concerning ciphertext expansion and low arithmetic intensity. The expansion rate, denoting the increase in ciphertext size compared to the original message, poses a significant concern. For instance, the BGV scheme, utilizing all available slots, exhibits a message expansion rate spanning three to five orders of magnitude. Furthermore, due to the substantial ciphertext size, only a fraction fits within GPU cache, leading to frequent memory accesses during computation. Compounded by FHE's low arithmetic intensity[11][30][1], characterized by less than one operation per byte, achieving data reuse becomes challenging.

This confluence of factors impedes the practical applicability of FHE, necessitating solutions to en-

hance its efficiency for real-world usage. While efforts have focused on augmenting cache size and memory bandwidth, the exploration of ciphertext compression remains relatively unexplored, save for considerations within network transmission contexts[42]. Compressing ciphertext presents unique hurdles, as it comprises pairs of matrices with elements represented as random numbers, defying traditional compression algorithms.

In this research, we propose a novel multi-level ciphertext compression approach, leveraging insights from application and algorithm knowledge to address these challenges. Our strategy encompasses two levels: first, by reconfiguring data layout to reduce bandwidth requirements and introducing specialized instructions for efficient data access; second, by dynamically generating one-half of the ciphertext based on seed for read-only or freshly encrypted data, thereby reducing storage demands. Additionally, for critical workloads like Deep Neural Networks, we expedite the costly ReLU activation function by omitting digits irrelevant to comparisons, accepting a minor error margin, which has negligible impact on machine learning tasks.

Through modeling and evaluation, we demonstrate the efficacy of our compression strategy, achieving notable speedups of 5.4× compared to 100x[30], on neural network workloads like CryptoDL[23]. Moreover, our approach compresses read-only ciphertext to only 4.6× of its original size, leading to substantial reductions in memory usage and marked acceleration in performance across FHE applications.

Background

Encryption Algorithm

The BGV scheme is an encryption method constructed from Ring Learning with Error (RLWE). Algorithm 1 outlines the steps to encrypt a plaintext. Initially, a plaintext in the form of a polyno-

Table 5.1: Homomorphic Operation Building Block of BGV Scheme.

HE Op	Description	Composing Kernel
HADD	Add two ciphertexts	Element-wise add
PADD	Add ciphertext with plaintext	Element-wise add to constant
PMULT	Multiply ciphertext with plaintext	Hadamard-Mult
HMULT	Multiply two ciphertexts	NTT, Hadamard-Mult, Conv, Element-wise Add
HROTATE	Circular rotate ciphertext	ForbeniusMap, NTT, Hada-Mult, Ele-Add, Conv

mial with coefficients under modulus q is required. Next, a polynomial $A(X)$ is sampled, where the polynomial is represented as a matrix due to its size. Subsequently, polynomial $B(X)$ is computed based on polynomial $A(X)$ and randomly generated polynomials. The output ciphertext consists of a pair of polynomials $A(X)$ and $B(X)$. Since the polynomial $A(X)$ is randomly generated from a seed, for fresh ciphertext, we can regenerate the polynomial from the seed.

Algorithm 1 RLWE-Encrypt

Input: $\mu(X) \in R_p$
sample $A(X) \leftarrow R_q$ and $E(X) \leftarrow \chi$ ▷ A(X) is randomly sampled from a seed
$B(X) = A(X).S(X) + \delta.\mu(X) + E(X) \mod R_q$
Output: $C(X) = (A(X), B(X))$

Homomorphic Operation

Table 5.1 enumerates the homomorphic operations natively supported by the BGV scheme. Notably, HMULT and HROTATE are complex operations composed of multiple expensive kernel operations such as NTT.

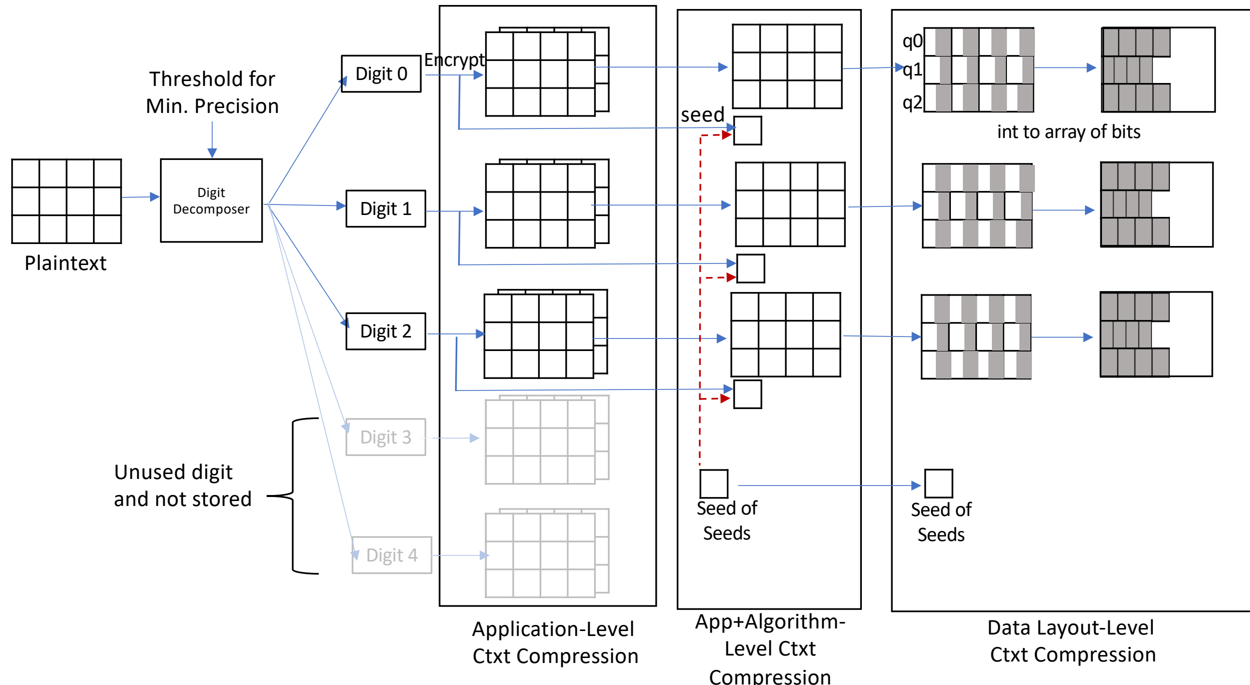

Figure 5.1: Three-level ciphertext compression based on application knowledge, algorithm knowledge, and data layout modification.

Architecture Design

Overall Architecture

The architecture for ciphertext compression is outlined in Figure 5.1, illustrating three stages of reduction in ciphertext usage.

Beginning with the leftmost rectangle, performing homomorphic comparison necessitates decomposing ciphertext/plaintext into multiple digits for subsequent comparison. Decomposition from ciphertext input yields multiple digits already in ciphertext format, while decomposition from plaintext input requires encryption of the resulting digit. The depicted output of the digit de-

composer includes five digits, yet only three are utilized for computation. The decomposer adjusts the number of digits based on an acceptable accuracy threshold determined beforehand, considering the degree of accuracy drop due to comparison errors. By disregarding certain digits, fewer ciphertexts are stored and compared, resulting in reduced computational load.

Moving to the second level, based on algorithmic insights, fresh or read-only ciphertext comprises a pair of matrices. Instead of retrieving the generated matrix from memory, it can be generated on-the-fly using a seed employed during encryption. An instruction supported by hardware facilitates pseudo-random number generation (PRNG), enabling storage of only half the original data size and saving memory bandwidth. Embedding seeds into application source code allows assignment of a single seed to each variable.

The final level of ciphertext compression involves modifying the data layout. Matrix elements, represented as integers under a chosen modulus, are typically smaller than 64 bits for performance reasons but large enough to meet precision requirements. Previous studies recommend a modulus of up to 36 bits for integers [34]. Storing data as arrays of bits rather than integers eliminates unused bits that occupy memory, thereby optimizing bandwidth usage. Since the data is stored as arrays of bits, accessing and utilizing it requires transforming the integer representation from arrays of bits. Given that FHE computations uniformly manipulate matrices, a new instruction for loading/storing ciphertext at a larger granularity is introduced, similar to the usage of matrix fragments in tensor cores.

Data Flow of a Ciphertext

As data is stored in memory as arrays of bits, conversion to a usable format is necessary. Figure 5.2 illustrates the process of fetching compressed data from memory to L1 cache for computation by the GPU core.

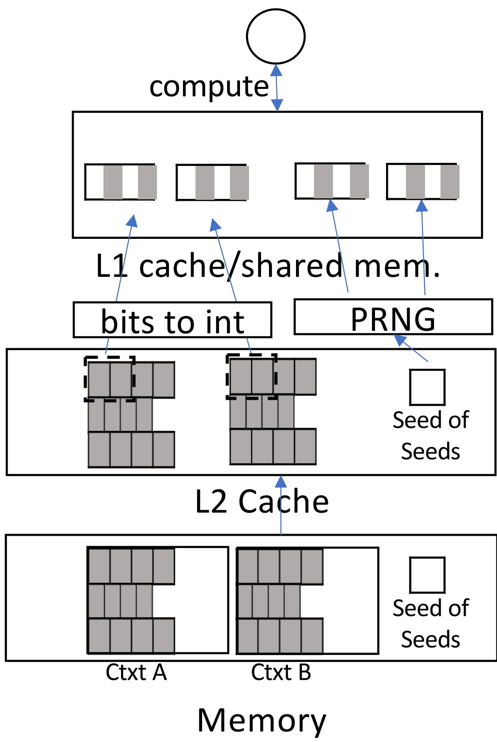

Figure 5.2: Data flow from memory to ALU (top) of a compressed ciphertext. Data fetched from memory to L2 cache is an array of bits. The converter module (bits to int) converts the array into integers. PRNG generates the read-only ciphertext part based on a seed.

The figure depicts the ciphertext being fetched from memory to L2 cache. For read-only ciphertext, one matrix is stored, and the other is generated using a seed. After fetching to L2 cache, a fragment of the ciphertext limb is retrieved and converted from arrays of bits to integers using a new instruction detailed in the subsequent subsection. The converted integers are then placed in L1 cache/shared memory for GPU core computation. For read-only data, part of the matrix is generated on-the-fly using PRNG, conserving memory bandwidth. PRNG implementation can be a GPU kernel in software or a hardware-backed PRNG engine. Opting for the latter reduces latency in matrix generation, hence the preference for a hardware-backed PRNG generator.

Reduced Precision of Comparison Operation

In FHE, performing a comparison operation necessitates decomposing numbers into multiple digits and comparing corresponding digits from two ciphertexts. For machine learning workloads, slight errors in these comparisons can be tolerated while maintaining acceptable inference accuracy. However, it's crucial to determine the extent of precision reduction that still yields acceptable results. To achieve maximum reduction while preserving accuracy, we employ a stepwise approach outlined in Figure 5.3.

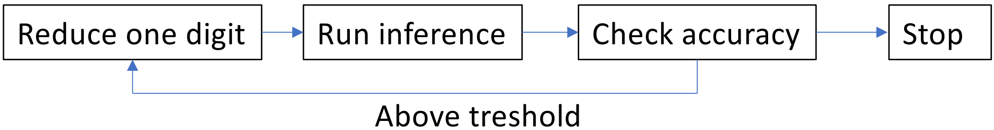

Figure 5.3: Steps to determine the minimum number of digits for acceptable accuracy.

Starting with the highest precision, we gradually reduce the number of digits in the comparison and assess if the resulting inference accuracy meets the minimum acceptable threshold. This iterative process continues until the optimal threshold is identified. Notably, reducing comparison precision decreases the number of ciphertexts involved, thereby improving execution time.

Load/Store Interface for Read-Only Ciphertext

In scenarios where the ciphertext exhibits a read-only access pattern, storing all ciphertext matrices becomes unnecessary. Instead, one matrix can be generated from a seed, while the other is stored in memory. We introduce new instructions for load/store operations to facilitate this generation, as depicted in Table 5.2.

This interface, inspired by matrix fragments used in Tensor Core operations for accelerating ma-

Table 5.2: Pseudo-Random Number Generator APIs

Variable declaration and interface for loading/storing PRNG generated matrix
wmma::fragment<unused, num_of_element, unused, unused, unused, unused> A_frag
wmma::load_limb_fragment_prng(A_frag, seed+index, modulus);
wmma::store_limb_fragment_prng(mem_destination, A_frag, modulus);

trix multiplication, enables efficient declaration and execution of load/store instructions. The num_of_element parameter in the variable declaration specifies the number of elements in the fragment limb to be generated. Operations are executed at the warp-level granularity. Given that the current FHE implementation fetches one element of a ciphertext matrix per thread, setting num_of_element to 32 allows fetching and generating 32 elements simultaneously, enhancing efficiency.

Load/Store Interface for Data Layout Modification

The matrix stored in memory is represented as an array of bits. To recover integers from this array, we introduce variable declarations and load/store APIs, as outlined in Table 5.3.

Table 5.3: Data Layout Converter APIs

Variable declaration and interface for loading/storing compressed ciphertext
wmma::fragment<unused, num_of_element, unused, unused, unused, unused> A_frag
wmma::load_limb_fragment(A_frag, mem_source, log2(modulus));
wmma::store_limb_fragment(mem_destination, A_frag, log2(modulus));

As depicted in the table, we begin by declaring a variable to hold the fetched matrix elements, specifying the number of elements to be fetched. Subsequently, the load/store APIs are utilized to retrieve the array of bits and transform the layout into 64-bit integers. This layout transformation is performed by dedicated hardware situated between the L1 and L2 cache. Notably, the number

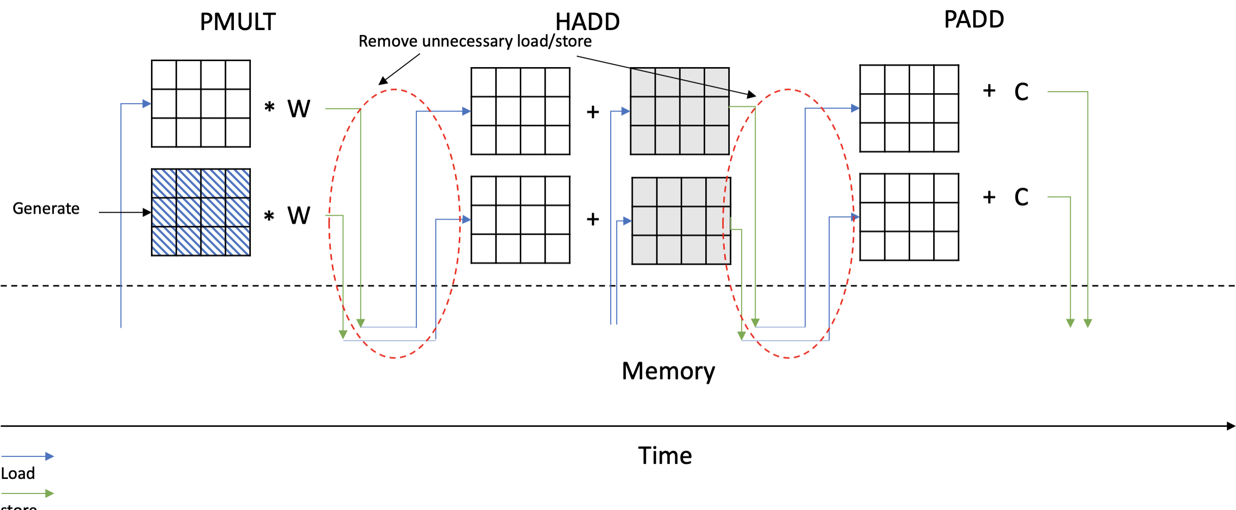

Figure 5.4: Timeline diagram depicting the matrix loads and stores for the sequential operations of PMULT, HADD, and PADD. Combining these operations will reduce the number of loads/stores for the matrix, as highlighted by the red oval lines.

of bits to be translated into a single 64-bit integer is specified as the last argument in the new load/store instructions.

Homomorphic Operation fusion and Lazy Operation

For PMULT, PADD, and HADD, these operations are bottlenecked by matrix loads and stores. For instance, HADD spends 67% of its time on matrix loads, 29% on matrix stores, and 4% on computation, kernel launch, and modulus data fetch. Therefore, reducing the number of loads/stores for these operations will significantly impact the operation latency. Figure 5.4 depicts a timeline diagram showing the sequential operations of PMULT, HADD, and PADD. These operations are typically found in the Convolutional and Fully Connected (FC) layers. As indicated by the red oval lines, by fusing these operations together, we can save some matrix loads and stores.

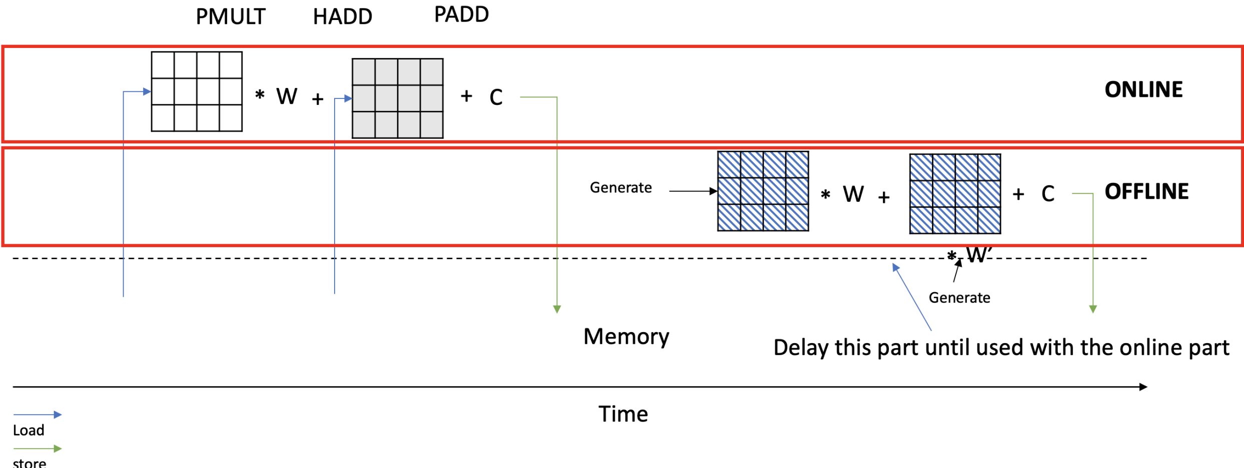

Figure 5.5: Timeline diagram illustrating the online and offline stages of the PMULT, HADD, and PADD operations, with the offline parts delayed to conserve matrix loads.

Table 5.4: GPU Configuration.

SM Configuration	82 SMs, 1395 MHz
Register File	256 KB/SM, 20.5 MB in total
Process Size	8 nm
L1D and Shared Mem.	128 KB
L2 cache	2 banks per memory partition, each L2 cache bank is 128 KB, 6 MB in total.
DRAM	24GB running at 1219 MHz, 24 partitions, 936.2 GB/s
PRNG engine	1 pipelined PRNG/SM with 40 cycles latency
Layout Converter	1 converter/SM with 5 cycles latency

Moreover, as depicted in Figure 5.5, we can divide the stages of the PMULT, HADD, and PADD operations into online and offline stages. The online stage involves loading matrices from memory, while the offline stage does not require matrix loads; instead, it is generated by a PRNG engine. Additionally, we can postpone the offline parts of the operation until the data is used, combining them with the online parts in HMULT operations. Delaying the offline part of the operation allows us to conserve matrix loads, thereby reducing the execution times of the operations.

Table 5.5: Practical FHE parameter used for homomorphic operation.

N	log(Q)	L	L_boot	dnum	λ
2^{16}	1728	23	17	3	128

Methodology

We employ Accel-sim [33] to model our scheme. The GPU configuration utilized to simulate the proposed architecture is detailed in Table 5.4, which mirrors the specifications of the Nvidia RTX 3090 based on the Ampere Architecture. Additionally, the table includes latency figures for the PRNG and the data converter from bits to integers. The simulation environment operates on Ubuntu OS version 18.04, NVIDIA driver version 525.105.17, CUDA version 12.0, and GCC version 8.4.0.

We configure the homomorphic operation for the BGV scheme using open-source GPU kernel code from [30]. Each homomorphic encryption is simulated until all GPU kernels have completed execution. We adopt FHE parameters conducive to bootstrapping, as outlined in Table 5.5, consistent with those used in [55]. We evaluate the performance of a Convolution Neural Network application employing the BGV scheme from CryptoDL [23] [64], where inputs are encrypted while weights remain unencrypted. The total execution times of the application are measured based on the latency of each homomorphic operation executed within the application.

Table 5.6: Homomorphic Operation Speedup over the Baseline [30].

HE Op	Layout	Layout+Prng	Layout+Prng+Lazy	TensorFHE
HADD	2.3×	3.2×	4.6×	2.4×
PADD	1.9×	2.4×	3.2×	2.3×
PMULT	1.9×	2.4×	3.2×	1.9×
HMULT	2.3×	2.5×	-	1.9×
HROTATE	2.4×	2.5×	-	1.9×

Evaluation

Homomorphic Operation Performance Speedup

First, we evaluate the speedup achieved for each homomorphic operation based on our scheme relative to the baseline of 100x [30], as outlined in Table 5.6. The layout modification, involving storing matrix elements as arrays of bits and fetching only one matrix of the ciphertext, significantly enhances execution times. Incorporating a PRNG provides further improvement, particularly for HADD, PADD, and PMULT, as these operations are heavily bottlenecked by matrix loads and stores. Lazy operation further enhances performance by eliminating matrix loads and stores during the offline stage through delayed operations. The last column demonstrates the speedup of TensorFHE[18] over the baseline. As observed, our achieved speedup with all optimizations surpasses the speedup achieved by TensorFHE for all operations.

Reduced Precision of Homomorphic Comparison

We conduct a comprehensive analysis of the error rate associated with performing homomorphic comparisons with reduced precision, as summarized in Table 5.7, utilizing the MNIST dataset. Initially, we perform comparisons with all digits (full precision) and progressively reduce the number

Table 5.7: Number of Homomorphic Operations for Each Comparison Operation with Different Precision and Error Rates.

HE Op	Full Precision (100%)	75%	50%	25%
HADD	103	80	57	18
PADD	15	12	9	6
PMULT	82	64	46	10
HMULT	45	34	23	12
HROTATE	5	5	5	5
Error rate	0%	1%	3%	10%

of digits compared until only 25% of the digits remain.

As depicted in the table, each comparison between ciphertexts involves multiple homomorphic operations. While comparing all digits ensures exact results, reducing the number of digits decreases computational overhead but introduces some comparison error. Remarkably, these errors remain tolerable within machine learning workloads, maintaining acceptable levels of inference accuracy.

Speedup on Neural Network Application

Figure 5.6 illustrates the speedup achieved by our schemes compared to TensorFHE relative to the baseline of 100x. As depicted in the figure, Our Work, which utilizes all optimizations except for comparison precision reduction, outperforms TensorFHE. Additionally, Our Work+ further improves the speedup; however, it comes with a reduction in inference accuracy.

To assess the efficacy of our optimizations in each homomorphic operation, we estimate the execution time achieved by the scheme and compare it against the baseline of 100x [30], as depicted in Figure 5.7. As illustrated in the figure, each optimization contributes to additional performance improvement. Furthermore, increasing the degree of precision reduction leads to higher speedups. Specifically, by utilizing only 25% of the digits for comparison, combined with other optimiza-

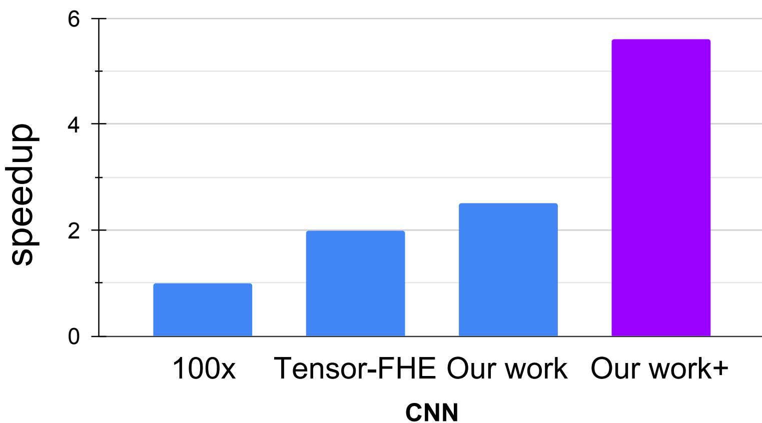

Figure 5.6: Comparison of the speedup achieved between TensorFHE, Our Work, and Our Work+ relative to the baseline (100x) for CNN applications.

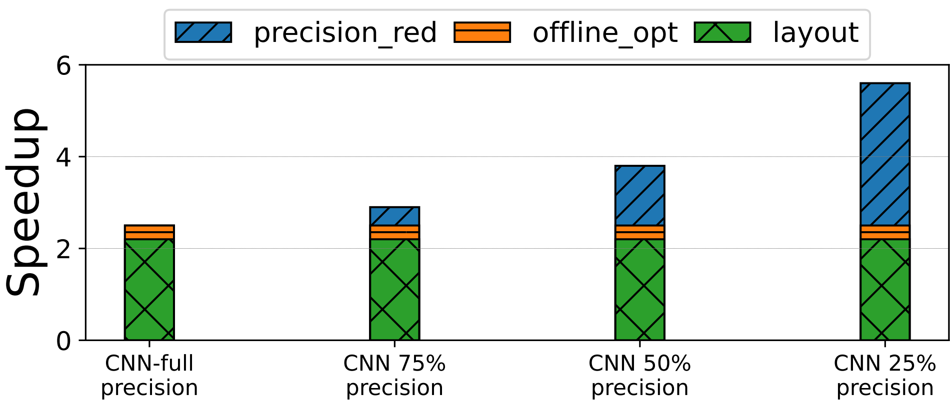

Figure 5.7: Speedup in execution time compared to the baseline for CNN application with varying degrees of precision reduction.

tions, we achieve a remarkable speedup of 5.6× over the baseline. Table 5.8 presents the accuracy of inference with varying comparison precision. As evident from the table, we achieve substantial execution time improvements with a mere 3% reduction in accuracy.

Table 5.8: The Accuracy of Neural Network Inference with Varying Comparison Precision.

Full precision (100%)	75%	50%	25%
97.94%	97.67%	97.35%	94.81%

Conclusion

In this study, we addressed the substantial computational demands of Fully Homomorphic Encryption (FHE) by implementing a three-level FHE computation memory usage reduction strategy derived from algorithmic insights, application-specific considerations, and data access patterns. Through evaluation using CNN workloads, we achieved a notable speedup of 5.6× with only approximately 3% accuracy reduction. This highlights the effectiveness of reducing memory accesses in highly memory-constrained workloads, leading to a significant acceleration of execution times.

CHAPTER 6: THESIS CONCLUSION

The GPU offers significant improvements in accelerating computation on non-confidential data. However, the computation of confidential data needs to be handled with special technology to ensure its confidentiality. This can be achieved either through a Trusted Execution Environment (TEE) or Fully Homomorphic Encryption (FHE) technology. Therefore, accelerating computations with GPUs requires a different approach compared to plaintext data processing.

In this thesis, we address the challenge of GPU acceleration for confidential computations using TEE or FHE. For TEE-based acceleration, we focus on the co-design problem discussed in Chapter 3. It's important to note that when CPU and GPU TEEs use different encryption modes, data re-encryption is necessary to match the encryption mode used at the destination. If the CPU and GPU TEEs are not co-designed and come from different manufacturers with varying hardware cycles, achieving this co-design can be complex. To overcome this, we propose the utilization of a software-based GPU TEE instead of a hardware-based one due to its flexibility. This approach allows for encryption mode adjustments in accordance with the CPU TEE even after manufacturing, avoiding the costly memory re-encryption process. However, managing the 128-bit data blocks required by AES encryption through the software layer presents challenges. We present three optimization strategies that effectively mitigate these challenges. With all three optimizations combined, we significantly reduce data collection and encryption overhead, making it negligible for regular benchmarks and acceptable (only 56% overhead) for irregular benchmarks.

For FHE-based confidential computation acceleration, especially focusing on comparison operations like "less than" and "equality," we target the BGV scheme, one of the most promising FHE schemes. The BGV scheme operates on integer data and operations, allowing a single ciphertext to store numerous input elements, enabling SIMD-style processing. While previous research has

explored GPU acceleration for FHE, the acceleration of comparison operations using an accelerator within the BGV scheme has been overlooked. In Chapter 4, we concentrate on accelerating BGV comparison operations through GPU and CPU parallelism. We identify and address several challenges related to offloading the most time-consuming parts of the FHE library to the GPU, such as dynamic memory allocation, buffer initialization, race conditions, and parallelizing sequential codes. Through our proposed solutions and optimizations, along with the inclusion of CPU parallelism, our approach achieves a noteworthy 11.1× speedup over the CPU-based FHE library.

Moreover, we demonstrate an alternative approach to mitigate the substantial computational overhead inherent in FHE applications: reducing the working memory footprint through ciphertext size compression, achieved by leveraging a combination of algorithmic and application-specific insights. This compression leads to smaller ciphertext size. Our investigation reveals that reducing precision for homomorphic comparisons combined with ciphertext compression results in a 5.6× performance improvement over the baseline on Convolution Neural Network inference workload.

Successfully addressing these challenges also opens up potential avenues for future research. The depth and breadth of future work can be expanded. In-depth investigations can focus on refining the solutions presented in this thesis to achieve lower overhead for TEE technology or greater speedup for the FHE approach. Additionally, widening the scope of this thesis could involve delving further into topics like enhancing integrity protection for GPU TEEs or exploring acceleration techniques for other components within the BGV bootstrapping mechanism.

APPENDIX A: COPYRIGHT INFORMATION

ACM Copyright and Audio/Video Release

Title of the Work: LITE: A Low-Cost Practical Inter-Operable GPU TEE
Submission ID:ics22-p24

Author/Presenter(s): Ardhi Wiratama Baskara Yudha:University of Central Florida;Jake Meyer:University of Central Florida;Shougang Yuan:North Carolina State University;Huiyang Zhou:North Carolina State University;Yan Solihin:University of Central Florida

Type of material:full paper

Publication and/or Conference Name: ICS '22: 2022 International Conference on Supercomputing Proceedings

I. **Copyright Transfer, Reserved Rights and Permitted Uses**

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's

employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work. (x) If your paper is withdrawn before it is published in the ACM Digital Library, the rights revert back to the author(s).

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new [ACM Consolidated TeX template Version 1.3 and above] automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

Please put the following LaTeX commands in the preamble of your document - i.e., before \begin{document}:

```
\copyrightyear{2022}
\acmYear{2022}
\setcopyright{acmcopyright}\acmConference[ICS '22]{2022 International Conference on Supercomputing}{June 28--30, 2022}{Virtual Event, USA}
\acmBooktitle{2022 International Conference on Supercomputing (ICS '22), June 28--30, 2022, Virtual Event, USA}
\acmPrice{15.00}
\acmDOI{10.1145/3524059.3532361}
\acmISBN{978-1-4503-9281-5/22/06}
```

NOTE: For authors using the ACM Microsoft Word Master Article Template and Publication Workflow, The ACM Publishing System (TAPS) will add the rights statement to your papers for you. Please check with your conference contact for information regarding submitting your source file(s) for processing.

If you are using the ACM Interim Microsoft Word template, or still using or older versions of the ACM SIGCHI template, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific

permission and/or a fee. Request permissions from Permissions@acm.org.

ICS '22, June 28–30, 2022, Virtual Event, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9281-5/22/06...$15.00
https://doi.org/10.1145/3524059.3532361

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country/region and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or

attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have any artistic images.

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

Funding Agents

1. NSF award number(s): 1908079
2. AMD award number(s):
3. NSF grant award number(s): 1908406
4. AMD gift fund award number(s):

DATE: **05/03/2022** sent to yudha@Knights.ucf.edu at **12:05:28**

LIST OF REFERENCES

[1] Rashmi Agrawal, Leo De Castro, Chiraag Juvekar, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. "MAD: Memory-Aware Design Techniques for Accelerating Fully Homomorphic Encryption". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, 685–697. ISBN: 9798400703294. DOI: 10.1145/3613424.3614302. URL: https://doi.org/10.1145/3613424.3614302.

[2] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. "HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data". In: *Privacy Enhancing Technology Symposium (PETs) 2023* (2023). URL: https://petsymposium.org/popets/2023/popets-2023-0020.php.

[3] Ehud Aharoni, Nir Drucker, and Hayim Shaul. "Advanced HE packing methods with applications to ML". In: *ACM Annual Conference on Computer and Communications Security*. 2022.

[4] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (Oct. 2015), pp. 169–203. ISSN: 1862-2984. DOI: 10.1515/jmc-2015-0016.

[5] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. *NVIDIA H100 Tensor Core GPU Architecture*. 2022. URL: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth.

[6] Ardhi Wiratama Baskara Yudha, Keiji Kimura, Huiyang Zhou, and Yan Solihin. "Scalable and Fast Lazy Persistency on GPUs". In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 2020, pp. 252–263. DOI: 10.1109/IISWC50251.2020.00032.

[7] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. "Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, 57–62. ISBN: 9781450386562. DOI: 10.1145/3474366.3486926. URL: https://doi.org/10.1145/3474366.3486926.

[8] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. *CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes*. Cryptology ePrint Archive, Paper 2018/758. https://eprint.iacr.org/2018/758. 2018. URL: https://eprint.iacr.org/2018/758.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, 309–325. ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262.

[10] CapePrivacy. *Cape Privacy: Privacy & trust management for machine learning*. https://capeprivacy.com/. 2021.

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

[12] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. "Pannotia: Understanding irregular GPGPU graph applications". In: *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 2013, pp. 185–195. DOI: 10.1109/IISWC.2013.6704684.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017*.

Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[14] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. *Efficient Homomorphic Comparison Methods with Optimal Complexity*. Cryptology ePrint Archive, Paper 2019/1234. `https://eprint.iacr.org/2019/1234`. 2019. URL: `https://eprint.iacr.org/2019/1234`.

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: *Advances in Cryptology – ASIACRYPT 2016*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 3–33. ISBN: 978-3-662-53887-6.

[16] Victor Costan and Srinivas Devadas. "Intel sgx explained." In: *IACR Cryptol. ePrint Arch.* 2016.86 (2016), pp. 1–118.

[17] DualityTechnologies. *DualityTechnologies: Data encryption technology and secure collaboration*. `https://dualitytech.com/`. 2022.

[18] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang. "TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 922–934. DOI: `10.1109/HPCA56546.2023.10071017`. URL: `https://doi.ieeecomputersociety.org/10.1109/HPCA56546.2023.10071017`.

[19] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Fully Homomorphic Encryption with Polylog Overhead". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 465–482. ISBN: 978-3-642-29011-4.

[20] Craig Gentry et al. "Fully homomorphic encryption using ideal lattices." In: *ACM Symposium on Theory of Computing*. Vol. 9. 2009, pp. 170–178.

[21] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. London, GBR: Samurai Media Limited, 2015. ISBN: 9789888381968.

[22] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481. 2020. URL: https://eprint.iacr.org/2020/1481.

[23] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. "Cryptodl: Deep neural networks over encrypted data". In: *arXiv preprint arXiv:1711.05189* (2017).

[24] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices". In: *IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007)* (2019), pp. 1–41. DOI: 10.1109/IEEESTD.2019.8637988.

[25] Ilia Iliashenko and Vincent Zucca. *Faster homomorphic comparison operations for BGV and BFV*. Cryptology ePrint Archive, Paper 2021/315. https://eprint.iacr.org/2021/315. 2021. URL: https://eprint.iacr.org/2021/315.

[26] Inpher. *Inpher: Secret computing and privacy-preserving analytics*. https://www.inpher.io/. 2022.

[27] Intel. *Intel® Architecture Memory Encryption Technologies Specification*. 2019. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/multi-key-total-memory-encryption-spec-753926.pdf.

[28] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. "Heterogeneous Isolated Execution for Commodity GPUs". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, 455–468. ISBN: 9781450362405. DOI: 10.1145/3297858.3304021. URL: https://doi.org/10.1145/3297858.3304021.

[29] Lei Jiang and Lei Ju. *FHEBench: Benchmarking Fully Homomorphic Encryption Schemes*. 2022. arXiv: 2203.00728 [cs.CR].

[30] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. "Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), 114–148. DOI: 10.46586/tches.v2021.i4.114-148. URL: https://tches.iacr.org/index.php/TCHES/article/view/9062.

[31] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. 2016. URL: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[32] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[33] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047.

[34] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. "SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400700958. DOI: 10.1145/3579371.3589053. URL: https://doi.org/10.1145/3579371.3589053.

[35] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022, pp. 1237–1254. DOI: 10.1109/MICRO56248.2022.00086.

[36] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. "BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, 711–725. ISBN: 9781450386104. DOI: 10.1145/3470496.3527415. URL: https://doi.org/10.1145/3470496.3527415.

[37] O. Kwon, Y. Kim, J. Huh, and H. Yoon. "ZeroKernel: Secure Context-isolated Execution on Commodity GPUs". In: *IEEE Transactions on Dependable and Secure Computing* (2019), pp. 1–1. DOI: 10.1109/TDSC.2019.2946250.

[38] Bingyao Li, Jieming Yin, Youtao Zhang, and Xulong Tang. "Improving Address Translation in Multi-GPUs via Sharing and Spilling Aware TLB Design". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, 1154–1168. ISBN: 9781450385572. DOI: 10.1145/3466752.3480083. URL: https://doi.org/10.1145/3466752.3480083.

[39] Yuan Lin and Vinod Grover. *Using CUDA Warp-Level Primitives*. 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives.

[40] Zhen Lin, Utkarsh Mathur, and Huiyang Zhou. "Scatter-and-gather revisited: High-performance side-channel-resistant AES on GPUs". In: *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*. 2019, pp. 2–11.

[41] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. "PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1057–1073.

[42] Rasoul Akhavan Mahdavi, Abdulrahman Diaa, and Florian Kerschbaum. "HE is all you need: Compressing FHE Ciphertexts using Additive HE". In: *arXiv preprint arXiv:2303.09043* (2023).

[43] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 2016, pp. 1–9.

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

[45] Seonjin Na, Sunho Lee, Yeonjae Kim, Jongse Park, and Jaehyuk Huh. "Common Counters: Compressed Encryption Counters for Secure GPU Memory". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 1–13. DOI: `10.1109/HPCA51647.2021.00011`.

[46] Ali Sah Ozcan, Can Ayduman, Enes Recep Turkoglu, and Erkay Savas. "Homomorphic Encryption on GPU". In: *IEEE Access* (2023), pp. 1–1. DOI: `10.1109/ACCESS.2023.3265583`.

[47] Ozgun Ozerk, Can Elgezen, Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. *Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2021/124. `https://eprint.iacr.org/2021/124`. 2021. URL: `https://eprint.iacr.org/2021/124`.

[48] NVIDIA Pascal. *NVIDIA Tesla P100 Whitepaper*. 2016.

[49] FIPS Pub. “Data encryption standard (DES)”. In: *FIPS PUB* (1999), pp. 46–3.

[50] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. “HEAX: An Architecture for Computing on Encrypted Data”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, 1295–1309. ISBN: 9781450371025. DOI: 10.1145/3373376.3378523. URL: https://doi.org/10.1145/3373376.3378523.

[51] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. “F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, 238–252. ISBN: 9781450385572. DOI: 10.1145/3466752.3480070. URL: https://doi.org/10.1145/3466752.3480070.

[52] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. “CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data”. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, 173–187. ISBN: 9781450386104. DOI: 10.1145/3470496.3527393. URL: https://doi.org/10.1145/3470496.3527393.

[53] Shiyu Shen, Hao Yang, Yu Liu, Zhe Liu, and Yunlei Zhao. “CARM: CUDA-Accelerated RNS Multiplication in Word-Wise Homomorphic Encryption Schemes for Internet of Things”. In: *IEEE Transactions on Computers* (2022), pp. 1–12. DOI: 10.1109/TC.2022.3227874.

[54] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. L. Abellan, J. Kim, and D. Kaeli. “Accelerating Polynomial Multiplication for Homomorphic Encryption on

GPUs". In: *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 2022, pp. 61–72.

[55] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. "GME: GPU-based Microarchitectural Extensions to Accelerate Homomorphic Encryption". In: *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2023).

[56] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 387–398. DOI: 10.1109/HPCA.2019.00052.

[57] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. "Parboil: A revised benchmark suite for scientific and commercial throughput computing". In: *Center for Reliable and High-Performance Computing* 127 (2012).

[58] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. "GPUvm: GPU Virtualization at the Hypervisor". In: *IEEE Transactions on Computers* 65.9 (2016), pp. 2752–2766. DOI: 10.1109/TC.2015.2506582.

[59] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. "GPUvm: Why Not Virtualizing GPUs at the Hypervisor?" In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 109–120. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki.

[60] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shuqin Ren, and Khin Mi Mi Aung. "Efficient Private Comparison Queries Over Encrypted Databases Using Fully

Homomorphic Encryption With Finite Fields”. In: *IEEE Transactions on Dependable and Secure Computing* 18.6 (2021), pp. 2861–2874. DOI: 10.1109/TDSC.2020.2967740.

[61] Nvidia Team. *CUDA C++ Programming Guide*. 2022. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[62] NVIDIA Tesla. *NVIDIA TESLA V100 GPU Architecture Whitepaper*. 2017.

[63] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. “Graviton: Trusted Execution Environments on GPUs”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 681–696. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/volos.

[64] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. “HE-Booster: An Efficient Polynomial Arithmetic Acceleration on GPUs for Fully Homomorphic Encryption”. In: *IEEE Transactions on Parallel and Distributed Systems* 34.4 (2023), pp. 1067–1081. DOI: 10.1109/TPDS.2022.3228628.

[65] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. “SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1483–1496.

[66] Chenyu Yan, D. Englender, M. Prvulovic, B. Rogers, and Yan Solihin. “Improving Cost, Performance, and Security of Memory Encryption and Authentication”. In: *33rd International Symposium on Computer Architecture (ISCA’06)*. 2006, pp. 179–190. DOI: 10.1109/ISCA.2006.22.

[67] Shougang Yuan, Ardhi Wiratama Baskara Yudha, Yan Solihin, and Huiyang Zhou. “Analyzing Secure Memory Architecture for GPUs”. In: *2021 IEEE International Symposium on*

Performance Analysis of Systems and Software (ISPASS). 2021, pp. 59–69. DOI: 10.1109/ISPASS51385.2021.00017.

[68] Shougang Yuan, Yan Solihin, and Huiyang Zhou. "PSSM: Achieving Secure Memory for GPUs with Partitioned and Sectored Security Metadata". In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, 139–151. ISBN: 9781450383356. DOI: 10.1145/3447818.3460374. URL: https://doi.org/10.1145/3447818.3460374.

[69] Ardhi Wiratama Baskara Yudha, Reza Pulungan, Henry Hoffmann, and Yan Solihin. "A Simple Cache Coherence Scheme for Integrated CPU-GPU Systems". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218664.

[70] Zama. *Concrete ML*. https://www.zama.ai/concrete-ml. 2022.

[71] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. "FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2023, pp. 896–907. DOI: 10.1109/HPCA56546.2023.10071133.