

Cerium: A Multi-GPU Framework for Terabyte-Scale Encrypted Inference

Siddharth Jayashankar
sidjay@cmu.edu
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Joshua Kim*
joshk@cs.utexas.edu
Computer Science Department
The University of Texas at Austin
Austin, Texas, USA

Michael B. Sullivan
misullivan@nvidia.com
Architecture Research Group
NVIDIA
Austin, Texas, USA

Wenting Zheng
wenting@cmu.edu
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Dimitrios Skarlatos
dskarlat@cs.cmu.edu
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

Encrypted AI using fully homomorphic encryption (FHE) enables inference directly over encrypted queries, providing strong privacy guarantees. However, its computational and memory overheads have limited practical deployment. Custom FHE accelerators improve performance, but rely on advanced manufacturing technologies that limit their accessibility. GPUs offer a more widely available alternative, yet achieving ASIC-class performance on GPUs is challenging. Large models such as LLMs compound these challenges by requiring optimized kernels, terabyte-scale memory management, and efficient execution across multiple devices.

We present Cerium, a multi-GPU framework for large-scale FHE inference. Cerium combines a domain-specific language, an optimizing compiler, and a runtime system to automatically generate optimized GPU kernels, compress plaintext data, manage terabyte-scale memory footprints, and optimize multi-GPU communication.

Implemented on NVIDIA GPUs, Cerium outperforms expert-written GPU libraries by up to 2.25× and achieves performance competitive with state-of-the-art FHE ASICs, matching the CraterLake ASIC. It is the first GPU system to execute CKKS bootstrapping in under 10 milliseconds, achieving 7.5 milliseconds, and is the first to demonstrate encrypted inference for BERT-Base and Llama3-8B prefill in 8 seconds and 43 seconds, respectively. The Cerium framework is open-source and available at https://github.com/CMU-CAOS/Cerium

CCS Concepts: • **Software and its engineering** → **Source code generation**; **Runtime environments**; • **Security and privacy** → **Cryptography**; • **Computing methodologies** → *Artificial intelligence.*

Keywords: Fully Homomorphic Encryption, GPUs, Compilers, Encrypted AI, Cryptography, Privacy Preserving Machine Learning

ACM Reference Format:
Siddharth Jayashankar, Joshua Kim, Michael B. Sullivan, Wenting Zheng, and Dimitrios Skarlatos. 2026. Cerium: A Multi-GPU Framework for Terabyte-Scale Encrypted Inference. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3830418.3843864

1 Introduction

Fully homomorphic encryption (FHE) enables computing directly on encrypted data, providing strong privacy guarantees for applications such as financial services and healthcare. In this paradigm, data remains encrypted throughout the computation and only the data owner can decrypt the results. FHE's confidentiality is mathematical and does not require trust in the hardware, software or any non-collusion assumptions. These properties make FHE attractive for secure inference, enabling users to perform AI inference over encrypted inputs on cloud-hosted models.

Unfortunately, the adoption of FHE in real-world applications has been limited by its massive performance overhead on high-end CPUs. To address this, prior efforts [28, 32, 44] proposed ASICs to accelerate FHE. However, they require cutting-edge technologies (e.g., high-end nodes, CoWoS [46], HBM) and entail high fabrication costs, making them currently impractical for widespread production and deployment. Given these constraints, datacenter GPUs emerge as a practical and accessible alternative to FHE ASICs. FHE programs are fundamentally dense vector computation and GPUs mirror several features required for FHE like massive parallelism, large on-chip caches, and high-bandwidth memory. However, attaining ASIC-like performance using GPUs

*Work done at Carnegie Mellon University.



This work is licensed under a Creative Commons Attribution International 4.0 License.

SOSP '26, Prague, Czech Republic
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843864

has remained elusive. As we will show in this paper, this is not because the hardware is inadequate, but because attaining ASIC-like performance requires a tightly integrated software stack that reasons holistically about FHE computation — creating optimized kernels, managing terabyte-scale memory and coordinating multi-GPU communication in ways that library-based approaches cannot achieve.

Prior works [3, 15, 16, 19, 21, 29, 30, 49] build GPU libraries for FHE that focus on kernels for low-level FHE primitives, however, they leave a substantial amount of performance untapped as FHE primitives map to very short-running kernels (5–10 us), leading to high kernel launch overheads and underutilizing hardware resources. State-of-the-art libraries like Cheddar [15] demonstrate significantly better speedups for small FHE workloads such as ResNet-20 by manually implementing kernel fusion for a limited set of FHE primitives. In general, achieving fast performance requires highly optimized kernels. However, manual optimization becomes tedious since every new application requires specialized optimizations across thousands of operations; and in practice, such difficulty leads to missed opportunities for optimization. Large models such as BERT-Base and Llama3-8B exacerbate the problem as they require optimized FHE kernels for operators such as attention, softmax, GELU, square roots, which are far more sophisticated than the univariate polynomials supported by prior works [15]. Consequently, achieving high-performance encrypted inference at scale requires a general, automated mechanism to generate optimized GPU kernels for arbitrary FHE programs.

Beyond kernel-level optimizations, memory management emerges as a central and unaddressed challenge. Existing GPU FHE work sidestep this issue as they focus on small models that fit entirely within a single GPU's memory. However, large FHE workloads like BERT-Base and Llama3-8B inference impose extreme memory pressure from three sources: (i) large encoded model weights, (ii) massive intermediate ciphertexts, and (iii) substantial evaluation-key sets. Together, they create enormous capacity requirements and heavy allocation overheads, quickly exhausting GPU and even host memory. Addressing this challenge fundamentally requires accounting for memory capacity, performing aggressive memory reuse, and host–device orchestration.

Finally, performance is fundamentally constrained by the limits of a single device. Closing the gap between FHE ASICs and GPUs ultimately requires multi-GPU scaling. This approach demands tightly coordinated execution, and efficient cross-GPU communication—capabilities absent from existing library-based efforts but essential for scaling FHE performance beyond the limits of a single device.

To address these challenges, this paper presents Cerium—a multi-GPU framework for terabyte-scale FHE workloads. Cerium consists of three integrated components: a domain-specific language, a compiler, and a runtime system. These components work synergistically to automatically generate optimized GPU kernels, manage memory for terabyte-scale workloads, and parallelize programs across multiple GPUs. Through this end-to-end design, Cerium enables running small and large FHE workloads with a performance that matches FHE ASICs for the first time.

To automatically generate optimized kernels, our insight is that FHE primitives map to a small set of well-structured kernels. This enables defining rule-based checks on the validity of fusing kernels which can be iteratively applied to the computation graph of the entire program. To accomplish this, Cerium introduces an IR, compiler passes, and heuristics to compile FHE circuit representations to high-performance GPU kernels, thus eliminating the need to painstakingly hand-optimize application-specific kernel implementations.

To address the memory capacity challenge, our primary insight is to take advantage of a symmetry in the algebraic structure of the FHE encoding algorithm to introduce an encoding that enables the compiler to implement 100× lossless memory compression. Additionally, we design a memory layout based on the heterogeneous nature of memory accesses in FHE to enable efficient host-device memory orchestration.

Finally, Cerium bridges the performance gap between GPUs and FHE ASICs by parallelizing FHE programs across multiple GPUs using compiler passes to reduce expensive inter-GPU communication operations, along with scheduling to overlap computation and communication.

We implement Cerium on NVIDIA GPUs and evaluate it on encrypted inference workloads of varying sizes. For small models, Cerium outperforms prior expert-written hand optimized GPU libraries by 1.25× on a single GPU and 2.25× with multi-GPU scaling. More importantly, Cerium's performance is competitive with recent state-of-the-art FHE ASICs, matching CraterLake [44], coming within 2.3× of ARK [32] and 4.4× of the multi-ASIC Cinnamon [28]. Cerium is the first to break the 10 ms barrier for a CKKS bootstrapping implementation, achieving 7.5 ms with off-the-shelf hardware. For large models, Cerium reduces memory capacity requirements by 100× and outperforms prior state-of-the-art GPU FHE BERT-Base inference [38] by 9.12× and demonstrates the first FHE Llama3-8B prefill with a 43-second TTFT.

Overall, Cerium makes the following contributions:

- An end-to-end multi-GPU framework for large-scale FHE machine learning inference
- A DSL and compiler for automatic generation of optimized GPU kernels for FHE programs
- A plaintext encoding that reduces memory capacity requirements for large models by over 100× by exploiting algebraic structure
- Compiler optimizations to reduce inter-GPU traffic by 44%
- A runtime system for kernel scheduling and memory management for efficient multi-GPU FHE orchestration

- An evaluation of the framework on small and large FHE workloads demonstrating that current GPUs can get within 4.4× the performance of state-of-the-art FHE ASICs
- The first to perform BERT-Base inference in 8.8 seconds and Llama3-8B prefill in 43 seconds, respectively
- Open Source: The Cerium framework is available at https://github.com/CMU-CAOS/Cerium

2 Background

In this section, we provide background on the threat model and FHE computation for encrypted AI inference using CKKS [13], a popular FHE scheme for computing on real-valued data.

2.1 Threat Model

We assume a threat model similar to that of prior two-party encrypted inference work [24, 36, 38, 50]. In this model, there are two parties: a user and a model service provider. We assume that the two parties are honest-but-curious. A user encrypts a query and sends the resulting ciphertexts to a server for inference. The server holds the model weights in plaintext and evaluates the model over the encrypted query. The user's query, and all values derived from it, receives cryptographic confidentiality without requiring trust in server-side hardware or software. We assume that the model architecture is public. Consequently, information exposed by the execution pattern, including the input size, number of generated tokens, identity or architecture of the queried model, and other metadata inferable from the size or structure of the computation, is outside the scope of our threat model. Furthermore, as the client receives outputs of the model inference, attacks such as model stealing are out of scope. Finally, FHE provides confidentiality but does not, by itself, guarantee the integrity or verifiability of the computation. Protecting against a malicious server that returns incorrect results is therefore also outside the scope.

2.2 CKKS Background

Figure 1 depicts the encryption steps in CKKS [13], an FHE scheme for encrypted computing on real values.

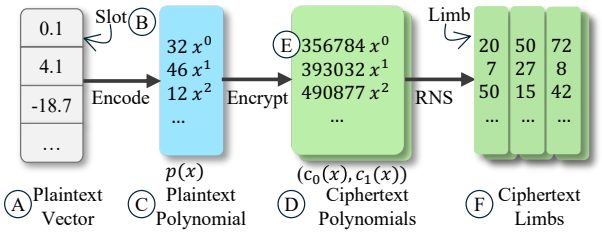


Figure 1. Encoding and Encrypting Values in RNS-CKKS

Slots, Encoding and Encryption. CKKS operates over modular polynomial rings, enabling batching vectors of real values Ⓐ into a single plaintext or ciphertext. Each entry in this vector is referred to as a slot Ⓑ. Encoding converts this vector into a plaintext polynomial Ⓒ and encryption converts it to a ciphertext Ⓓ, which is a pair of modular polynomials.

Homomorphic Operations. CKKS supports three primitive homomorphic operations: (i) addition, (ii) multiplication, and (iii) cyclic rotation, that operate slotwise on the encrypted plaintext vector. All higher-level computation is expressed as circuits composed of these primitives. This restrictive model has two major implications for ML workloads. First, CKKS cannot natively express any nonlinear function. Therefore nonlinearities like division, ReLU, max, softmax, SiLU, etc., require the use of polynomial approximations whose degree and precision must be tailored to model accuracy requirements. Furthermore, CKKS lacks a direct indexing operator for accessing individual slots and thus cannot natively support tensor operations. Instead, tensor values have to be "packed" into slots and operations implemented using combinations of cyclic rotations, multiplications, and additions. This makes circuit implementations of tensor operations sensitive to factors like layer dimensions, convolution strides, batch sizes, token counts etc. As a result, FHE circuits for ML models lack generality and are tightly coupled to specific parameters making it hard to hand craft optimized implementations for all these cases.

RNS and Limbs. Polynomial coefficients Ⓔ are very large integers, often thousands of bits. Performing modular arithmetic directly on such values is inefficient. In practice, the ciphertext modulus is chosen as the product of a set of machine word-sized primes and the Residue Number System (RNS) [5] is used to decompose modular arithmetic over this large modulus into modular arithmetic over its prime factors, which form the RNS basis. Limbs Ⓕ refer to the modular residues of a polynomial over the RNS basis. Operations over limbs are the building blocks of practical CKKS implementations, and as most limb operations are data parallel in nature, RNS-CKKS becomes suitable for GPU acceleration.

NTT. The Number Theoretic Transform (NTT) is the analog of the Fast Fourier Transform (FFT) in a prime number field. Similar to FFT, NTT performs a convolution to speed up modular limb polynomial multiplication by transforming it from the coefficient domain to the evaluation domain. The inverse NTT (INTT) reverses this transformation. By default, all limbs can be assumed to be in the evaluation domain.

Bootstrapping. Every ciphertext has an intrinsic finite multiplicative budget that limits operations. Bootstrapping is an expensive ciphertext maintenance operation that refreshes the multiplicative budget of a ciphertext, enabling unbounded computation. Since bootstrapping is fundamental to all FHE computation, it is vital that it runs as fast as possible.

Evaluation Keys. Evaluation keys (evalkeys) are auxiliary data required by the keyswitching ciphertext maintenance operation. The total size of all evaluation keys in ML workloads is typically on the order of 10–50 GB.

3 Motivation

In this section, we motivate the need for a holistic compiler and runtime-based approach for FHE programming on GPUs by outlining four critical, intertwined challenges that manual or library-based methods cannot adequately solve.

3.1 Optimized Kernels are Application Specific

FHE programs are expressed as circuits of homomorphic operations over ciphertexts. While this abstraction is meaningful for reasoning about the program, it is not the appropriate granularity for an optimized GPU implementation as FHE primitives map to short-running kernels (5-10 us) and thus severely under-utilize the GPU due to limited parallelism for latency hiding, excessive inter-kernel data communication and, kernel launching overheads.

Achieving high performance requires combining multiple operations into larger "fused" kernels to reduce memory transfers and better utilize GPU hardware [15]. However, this process is extremely tedious and requires deep expertise in both GPU and FHE programming. Fundamentally, the fusion implementation is a function of the computation graph of kernels and extensive fusion results in the fused kernels becoming application-specific, and thus has to be repeated for every new application.

Cheddar [15] manually developed a limited set of fused kernel for patterns that are mostly tailored to small workloads. For instance, Cheddar has limited support for fusion within a univariate polynomial and lacks support for fusion across multiple polynomials. Large workloads like BERT-Base and Llama3-8B contain operations like ciphertext - ciphertext Matmul, softmax, square root, GELU that are far more complex and not supported by Cheddar. While the complexity opens the door to more fusion opportunities; it also makes manual fusion difficult to perform.

Resolving this problem requires a compiler that can automatically generate optimized kernels from FHE circuit specifications, performing domain-specific optimizations and fusion while also enabling generality across applications.

3.2 Encrypted AI Dramatically Stresses Memory Capacity and Management

FHE dramatically increases memory requirements, making memory management a first-order concern. Repeated runtime allocation and deallocation can introduce overheads of up to 100%, degrading performance even for small workloads. Memory capacity is an additional challenge for encrypted AI because model weights must be packed into plaintext polynomials and represented in RNS form, resulting in a multiplicative increase in their memory footprint.

Figure 2 shows the memory capacity requirements of encrypted inference for models of varying sizes. Even smaller models like ResNet-50 and MobileNet have memory footprints exceeding several hundreds of gigabytes [37], much

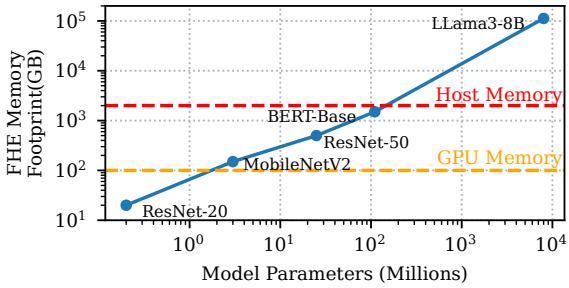


Figure 2. FHE Memory Footprint vs Model Parameters

larger than what can fit in a GPU's memory. Encrypted LLM inference like BERT-Base or Llama3-8B requires terabytes of memory and becomes far too large to fit in GPU memories and even host main memories. BERT-Base, for instance, needs 1.5 TB of memory, and Llama3-8B requires 112 TB. Beyond inputs, encrypted LLMs further contain large amounts of intermediate ciphertext values that further stress the memory capacity on the GPU. Previous GPU-FHE works [15, 19, 29, 30] focus on small models such as ResNet-20 and therefore avoid memory capacity challenges, or estimate end-to-end inference runtime by adding the runtime of each component [38, 50].

Efficient execution, therefore, requires memory management and orchestration of data movement between the host and GPUs. These requirements are well-suited to a compiler and runtime system that can perform memory management tasks such as liveness analysis, reuse memory buffers, and orchestrate data movement between host and GPUs.

3.3 Kernel Launch Overheads are Significant

FHE programs, even when driven by aggressively hand-optimized kernels, still suffer from substantial performance overheads rooted in kernel launch latency from online host to device interaction. Addressing these overheads requires asynchronous programming techniques that specify the computation as a graph of kernels that can be launched on the GPU in a single operation. However, manually creating such graphs, e.g., through CudaGraphs, is difficult because it requires precise tracking of kernel boundaries, memory lifetimes, and aliasing to avoid data races. As a result, this becomes a dynamic and cross-cutting task that is best handled by a compiler that can analyze dependencies among kernels to generate correct, modular, and reusable execution graphs.

3.4 Multi-GPU FHE Requires Fine-Grained Scheduling and Optimization

Parallelizing FHE programs across multiple GPUs can close the performance gap with ASICs. However, this is difficult as parallelism techniques like limb- and program-level parallelism [28] require minimizing costly inter-GPU communication, reordering and batching of communication operations, fine-grained scheduling to overlap communication with compute, and runtime management of CUDA streams

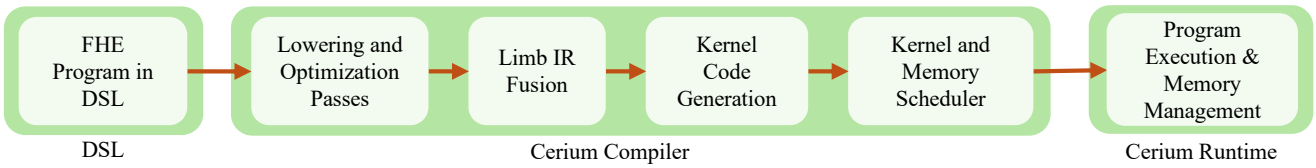


Figure 3. An Overview of the Cerium Framework.

and memory buffers to yield meaningful speedups. These tasks mandate a compiler and a runtime system that can orchestrate parallel FHE execution across-GPUs.

4 Design

Cerium is a multi-GPU framework for encrypted AI based on CKKS-FHE which supports the encryption of real values. The Cerium framework consists of three components: (i) a Domain-Specific Language (DSL), (ii) an optimizing compiler, and (iii) a runtime system.

Cerium's design philosophy is to separate application logic from system-level concerns, allowing FHE programmers to express computations without manually constructing high-performance GPU implementations. The framework automatically generates optimized GPU kernels, manages memory for terabyte-scale encrypted workloads such as Llama3-8B, and parallelizes FHE programs across multiple GPUs to narrow the performance gap with FHE ASICs.

Figure 3 presents a high-level view of the Cerium workflow. Programs are written using Cerium's Python embedded DSL, after which the compiler lowers them to polynomial and then limb level intermediate representations, applying optimizations at each stage. The compiler then groups limb-level IR operations into fused limb IR instructions, which are passed to the kernel code generator to produce optimized GPU kernels. Next, the compiler constructs a kernel and memory schedule that encodes data dependencies, kernel ordering, and memory allocations. The fused kernel code is compiled into a shared library and, together with the kernel and memory schedule, is provided to the Cerium runtime. The runtime initializes the program, launches kernels according to the schedule, manages host and GPU memory, and orchestrates inter-GPU communication. The following sections provide details on Cerium's design, outlining architectural choices, challenges encountered, and technical mechanisms developed to resolve them.

4.1 Domain-Specific Language

The Cerium DSL is a lightweight Python-embedded DSL that enables developers to express computations at the FHE circuit level, without dealing with low-level C++ or CUDA code. It supports static type checking to reduce programming errors and modular program decomposition through the `CeriumFunction` construct. Figure 4 shows an example implementing matrix multiplication using the Baby-Step Giant-Step [11] algorithm in the DSL. The language allows

```
CeriumFunction(name="matmul",gpus=4):
  x = CiphertextArg(name="x",level=20,scale=40)
  babySteps, giantSteps = [0..16], [16,32..256]
  mat = getBabyStepGiantStepPlaintexts("W",x.level(),
      scale=40,babySteps,giantSteps,period=256)
  prod = BabyStepGiantStep(x,mat,babySteps,giantSteps)
  prod = prod.doubleRescale()
  FunctionOutput(prod)

CeriumFunction(name="main",gpus=4):
  x = CiphertextInput(name="x",level=20,scale=40)
  A = CeriumFunctionCall("matmul",x,plaintexts="Wa")
  B = CeriumFunctionCall("matmul",x,plaintexts="Wb")
```

Figure 4. A Code Sample with the Cerium DSL.

declaring plaintext types with a `period` argument, which the compiler uses to implement the plaintext compression technique that will be introduced in Section 4.5. It supports multi-GPU parallelization using the limb and program parallelism techniques from Cinnamon [28]. Crucially, the DSL hides system-level details such as memory management, threading, kernel scheduling, and cross-GPU communication, delegating them to the compiler and runtime.

4.2 Polynomial and Limb IR Lowering

The Cerium compiler lowers programs in the DSL first to a polynomial-level IR and then to a limb-level IR, performing optimizations at each stage. The polynomial level IR is an extension of the IR in [28]. Here, we make use of optimizations from prior work like hoisting [26], double-hoisting [7], mod down merging [2], double rescale [31], limb-level parallelism [28]. After optimization, the representation is input to the limb IR fusion passes that group instructions together for eventual kernel creation.

4.3 Limb IR Fusion

The primary optimization for achieving high-performance FHE execution on GPUs is kernel fusion [19, 21, 30]. Prior works perform this process manually, in an ad hoc manner. In contrast, Cerium fully automates kernel fusion, enabling the systematic generation of high performance GPU kernels directly from FHE circuits.

The challenge in designing an automated kernel fusion technique lies in efficiently reasoning about the correctness and performance of the fused kernels. Considering performance is necessary, as kernel fusion is not universally beneficial. For example, kernel fusion stresses resources like register files and shared memory, and excessive fusion can

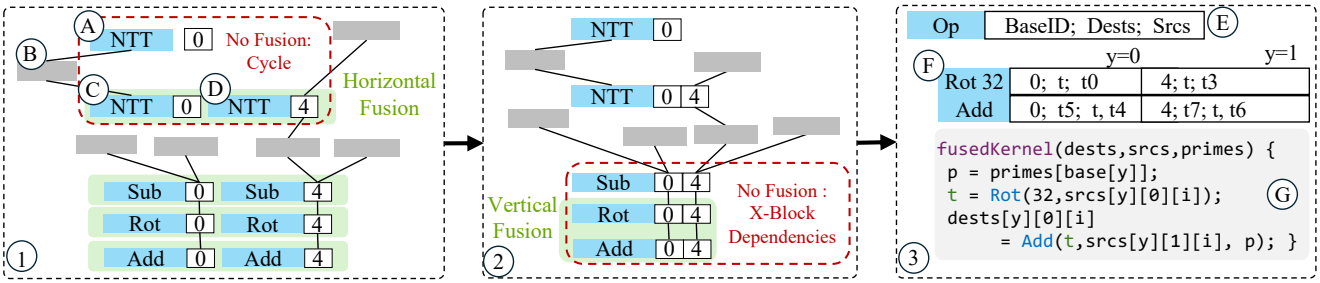


Figure 5. Limb IR Fusion and Code Generation

lead to poor performance. In general, automatically reasoning about kernel fusion is difficult as the search space and possibilities are very large.

To simplify the search space and enable efficient reasoning, our insight is to automate fusion at the limb level. Limbs are well suited for this task because limb operations (i) constitute the atomic building blocks of RNS-CKKS computation and (ii) can be partitioned into a small set of classes that map to fundamentally different kernel structures, which are not amenable to cross-class fusion. Since the set of limb operations and classes is small, the combinatorial possibilities of interactions between them is limited and can be explored using a rule-based approach over the limb operations.

Cerium introduces Limb IR, an intermediate representation that captures fundamental limb-level operations and abstracts their kernel implementations. Limb IR instructions consist of an opcode, RNS Base ID, and destination and source operands. They provide a compact abstraction of the underlying kernels, allowing the compiler to efficiently reason about correctness, dependencies, and resource utilization.

Cerium's kernel fusion is split into two passes. Cerium begins by performing horizontal fusion, i.e., fusion of data-independent limb IR operations into the same kernel. Horizontally fused instructions run in separate thread blocks of the materialized kernel. The primary benefit of this fusion is that it allows for submitting larger amounts of work to the GPU, increasing utilization and amortizing launch overheads. ① in Figure 5 depicts an example of horizontal fusion. It shows a DAG of unfused limb IR instructions. For brevity, we omit showing the source and destination operands as they can be inferred from graph edges. The digits indicate the RNS Base IDs of the limb IR instruction. For example, NTT4 indicates an NTT operation modulo the RNS prime at index 4. The gray boxes represent other limb IR instructions in the graph and are presented here to highlight dependencies between instructions.

The Cerium compiler iterates over the DAG of limb IR instructions, greedily checking for eligible horizontal fusion candidates in a pairwise manner. Limb IR instructions can be horizontally fused if they have the same opcode. For example, NTT instructions Ⓒ and Ⓓ can be fused together. However, fusion is disallowed if it results in a cycle in the graph. Thus, NTT instruction Ⓐ cannot be fused with instruction Ⓒ as fusion would create a cyclical dependency on Ⓑ. Horizontal fusion across different opcodes is disallowed as it would create control flow divergence in the generated GPU code. Limb IR instructions are annotated with opcode relevant metadata like level, base conversion input and output dimensions, mod down factors. Cerium employs rules to ensure that the metadata factors of the two limb IR instructions that are fused with each other are the same.

Following horizontal fusion, Cerium performs vertical fusion greedily. Vertical fusion combines instructions that contain data dependencies within the same thread, and thus improves performance by using registers to communicate data rather than expensive global memory. Vertical fusion requires checking for cyclical dependencies, Base ID alignment, and cross thread block data dependencies, as sharing data across thread blocks is unsafe. The Cerium compiler reasons about cross-block dependencies using the compatibility of opcodes and operand sources. ② in Figure 5 depicts an example of vertical fusion. Here, the (horizontally fused) rotate and add instructions are valid vertical fusion candidates as they have compatible opcodes and aligned RNS base IDs. However, the subtract instruction cannot be vertically fused with them, as this would result in both the source and destination operands of the rotate instruction residing in the same kernel, leading to unsafe cross-thread block communication from the permutation of values in the rotate instruction. For simplicity, Cerium limits vertical fusion to elementwise operations.

Compilation efficiency requires efficiently determining whether fusion will create cyclical dependencies. To do so, Cerium stores the parents and children of each limb IR instruction in the graph and checks whether the intersection between the parents of one instruction and the children of the other is empty, as a non-empty set implies a cycle. Furthermore, to keep the DAG size small, Cerium performs fusion within individual `CeriumFunctions` defined in the DSL program. However, if a function's limb IR DAG is too large, the compiler partitions it into smaller sub-DAGs and performs fusion independently within each. The sub-DAG

size is a configurable parameter that controls the trade-off between optimization aggressiveness and compilation latency. We experimentally pick the default sub-DAG size.

As we show in Sections 6.4 and 6.5, Cerium can compile large programs in a few minutes and its kernel fusion results in an average 2.87× performance improvement over an unfused, direct polynomial to kernel baseline and a 1.21× improvement over Cheddar [15].

4.4 Kernel Code Generation

After the limb IR fusion pass, the Cerium code generator lowers the abstract fused limb IR representation of the kernels into kernel code, as shown by ③ in Figure 5. Ⓔ depicts the format of a limb IR instruction consisting of an opcode, RNS base ID, and destination and source operands. Ⓕ shows a horizontally and vertically fused limb IR instruction. Ⓖ depicts the generated kernel (pseudo) code for Ⓕ. Here, `y` and `i` correspond to the index of horizontal fusion and index within the limb vector, respectively. We describe two heuristics used in Cerium's code generator to improve efficiency of generated kernels.

4.4.1 Load Reuse. As Elementwise kernels are bandwidth bound, we generate code that optimizes for reuse of values. When a value is a common input to different horizontally fused limb IR instructions with the same RNS base, the Cerium code generator will downgrade the horizontal fusion and place the values in the same thread. This reduces the overall bandwidth consumption, even though it comes at the cost of other factors — occupancy, latency, and parallelism.

4.4.2 Kernel Splitting. Unrestricted, vertically fused kernels can span thousands of instructions, significantly increasing the number of operands. This places pressure on the register file and may cause costly spills. For example, large matrix multiplication kernels contain thousands of plaintext–ciphertext multiply–accumulate operations. Figure 6 shows how the performance of ciphertext–plaintext matrix multiplication varies with the extent of vertical fusion, controlled by a threshold parameter which is a weighted sum of the count of the limb IR instructions in the kernel. Initially, increasing fusion provides substantial performance gains. However, the benefits eventually plateau as fusing large kernels yield diminishing returns. Beyond a point, excessive fusion increases register pressure, leading to expensive spills and steep performance degradation. For instance, a parameter value of 2K results in an aggregate of 70 KB of spills per thread across all kernels, causing a 2× performance drop. To avoid this pitfall, Cerium enforces a threshold on kernel size and splits kernels that exceed this limit into smaller kernels.

Similarly, excessive horizontal fusion stresses memory capacity and bandwidth as memory locations cannot be reused across thread blocks. Cerium estimates the number of memory locations required by a kernel and splits large horizontally fused kernels when required.

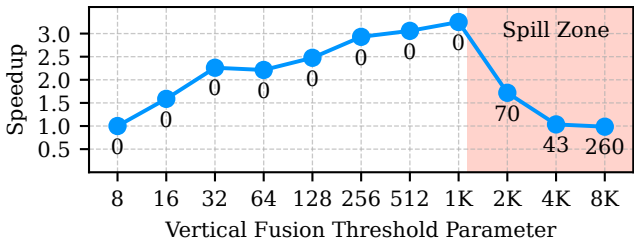


Figure 6. Speedup vs Vertical Fusion Threshold for a 128 × 4096 Ciphertext Matrix with a 4096 × 1024 Plaintext Matrix. Labels indicate aggregate kilobytes spilled per thread

4.5 Plaintext Compressed Encoding

As discussed in Section 3.2, FHE substantially increases memory requirements. This overhead is amplified by common packing schemes that repeat matrix values within plaintext vectors to reduce rotation costs [10, 11]. The resulting repetition increases both the number and total size of the required plaintexts. This is particularly problematic for encrypted inference with models such as BERT-Base and Llama3-8B, whose large weight matrices produce plaintext storage footprints of up to 100× their original size: 1.5 TB for BERT-Base and 112 TB for Llama3-8B.

To conserve memory, we introduce a compression technique that exploits the algebraic interaction between power-of-2 periodicity and the CKKS encoding algorithm to reduce the memory capacity required for plaintexts. The CKKS encoding algorithm first converts a plaintext vector in the time domain to a polynomial in the frequency domain by performing an inverse FFT. As a periodic vector has fewer independent frequencies, the resulting polynomial is sparse. When this sparse polynomial is transformed back to the (bit-reversed) NTT representation, it results in contiguous, repeated values. This contiguous repetition can be compressed by storing only distinct elements and tracking their original indices. An example is shown in Figure 7. Theorem 1 formally states this property. A full proof, along with constructions of ciphertext addition, multiplication, and matrix multiplication is available in Appendix A.

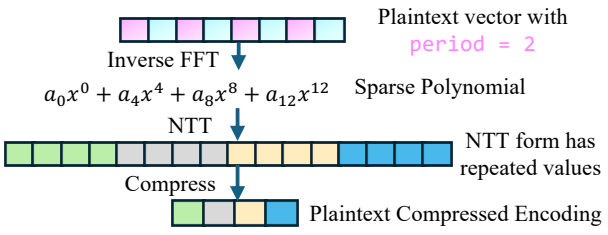


Figure 7. Plaintext Compressed Encoding

Theorem 1. *Let P be an M-slot vector of period T, where $T < M$ are powers of two. Under the permutation $k(i) = \frac{(5^i \mod 4M) - 1}{4}$, conjugate extension, IFFT, NTT, and bit-reversal, the resulting NTT representation, P_{ntt} has a contiguous layout with at most $2T$ distinct elements.*

The Cerium DSL supports declaring plaintext inputs with a power of 2 `period` argument. The compiler represents them in the compressed plaintext encoding and performs the necessary indexing transformations when generating code. As Section 6.5.4 will show, this optimization yields dramatic memory savings of more than 100×, enabling running LLMs like BERT-Base and Llama3-8B in FHE. Therefore, efficiently implementing large models in FHE requires creating packing strategies where the period is a power of two, enabling use of Cerium's compressed plaintext encoding for memory compression and performance.

4.6 Kernel and Memory Schedule Creation

To assemble the generated kernels into an executable program, the Cerium compiler constructs a kernel and memory schedule. This schedule contains information on the input and output operands and auxiliary data for each fused kernel. It also contains data dependencies between kernels and serves as the execution plan for the Cerium runtime, which orchestrates kernel launches and manages data flow.

When creating the execution plan, intermediate ciphertext values are a significant source of memory pressure, as the total sizes of operands can exceed hundreds of GB. To address this, Cerium reuses memory locations, and performs operations in-place, conserving valuable GPU memory and eliminating costly memory management. The challenge here is to reason about the correctness of the reuse and prevent data races; Cerium does so by efficiently reasoning about kernel data access patterns using the fused limb IR representation of the kernels. Fused limb IR instructions begin in the static single assignment form and are topologically ordered. The compiler determines the lifetime of values and assigns address locations. If the compiler determines that a value dies in a kernel, it attempts to recycle its location. For safe reuse, the compiler only reuses memory locations within a kernel if it can ensure that the dead value is read by only one thread in the kernel. This enables the compiler to reuse its address for a subsequent write in the same thread. This is valid as instructions within a thread are always executed in order. Rotation operands are restricted from being reused within the same kernel as thread ordering across blocks is not deterministic. Dead locations can be safely recycled across kernel boundaries as all reads are guaranteed to have completed before subsequent writes in the next kernel.

A second challenge is that FHE workloads exhibit heterogeneous data lifetimes. Some data persists across programs, while others change frequently. For example, evaluation keys (evalkeys) remain constant during a single inference session but differ across sessions with new keys. Plaintext weight matrices, on the other hand, are reused across inferences within the same layer but differ between layers, whereas other plaintexts, such as bootstrapping matrices or masks, are shared across both layers and inferences. Managing these distinct lifetimes efficiently requires a carefully designed memory layout and swapping strategy.

To handle these heterogeneous workload lifetimes, the Cerium compiler uses information about the type of values to create a memory layout that organizes GPU memory into dedicated memory regions for different data categories: plaintext weights, evalkeys, plaintexts, ciphertext inputs, and intermediate ciphertext values. Each pool is statically allocated and managed independently, allowing values to be updated or swapped simply by adjusting the pointer to the corresponding memory pool. This design simplifies data reuse across program executions. We revisit this design in Section 4.8 and show how it plays a crucial role in enabling memory and kernel scheduling optimizations that are performed by the Cerium runtime. These optimizations conserve valuable GPU memory and enable efficient host-to-GPU data movement for workloads like Llama3-8B that require more memory than is available on the GPU.

4.7 Multi-GPU Parallelization

As FHE programs are limited by memory capacity, bandwidth, and SM count, parallelizing execution across multiple GPUs can improve performance by increasing aggregate resources. The main challenge, however, lies in inter-GPU communication overhead. Specifically, each communication operation requires synchronization across GPUs and typically lasts several microseconds, significantly impacting end-to-end performance. To optimize FHE programs for multi-GPU execution, Cerium implements parallelization with two objectives: (i) minimizing inter-GPU communication and, (ii) scheduling communication operations to avoid idling and overlap them with computation.

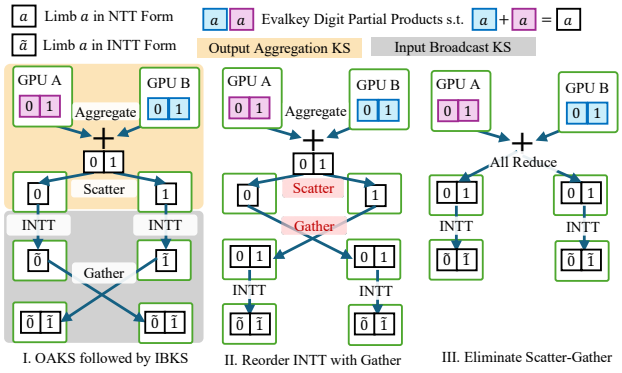


Figure 8. Merge aggregate scatter and all reduce across output aggregation and input broadcast keyswitches

4.7.1 Minimizing Inter-GPU Communication.

Cinnamon [28] identifies limb- and program-level parallelism in FHE and proposes two key-switching schemes: input broadcast key-switching (IBKS) and output aggregation key- switching (OAKS), each with distinct communication patterns. Cerium

builds on this by introducing compiler optimizations to further reduce communication overhead.

IBKS broadcasts the limbs to all chips via an all-gather operation (AG) at the beginning, while OAKS performs an aggregate-and-scatter (AS) at the end. When the output of OAKS serves as an input to an IBKS, Cerium fuses these steps into a single all-reduce by reordering operations, eliminating redundant scatter/gather phases.

Figure 8 depicts this optimization. ① shows the AS-step of an OAKS, followed by an IBKS on a 2 limb polynomial. As gather is only a data movement operation, the sequence: AS → INTT → AG can be reordered as AS → AG → INTT as shown in ②. Finally, as the scatter and gather operations cancel each other out, the communication operations can be simplified to a single all reduce (③). This transformation trades minor extra computation for significantly reduced communication and is commonly found as a result of successive matrix multiplication operations as is in bootstrapping.

Additionally, Cerium removes redundant broadcasts by replacing multiple broadcast-then-accumulate sequences with a single aggregate-scatter, reducing communication calls and synchronization.

4.7.2 Scheduling Computation and Communication.

To further hide communication latency, Cerium incorporates compiler transformations that reorder communication operations to occur as early as possible, allowing data transfers to proceed concurrently with downstream computation. Additionally, Cerium partitions long communication operations into smaller segments, enabling finer-grained interleaving of compute and communication, thereby preventing computation from waiting for a long-running communication operation to complete. At runtime, communication operations are dispatched to dedicated CUDA streams, distinct from those used for computation, allowing both phases to progress simultaneously. This overlap strategy hides communication latency and ensures that available GPU resources remain fully utilized.

4.8 Cerium Runtime

The kernels generated in Section 4.4 are compiled into a shared library and together with the kernel and the memory schedule from Section 4.6 are input to the Cerium runtime, which manages program execution. It initializes the program, links all compiled `CeriumFunctions` together, sets up inputs, allocates and manages memory, and handles kernel launching. In this section, we describe some of the key features of the Cerium runtime.

4.8.1 Memory Pool Reuse.

For large programs, the intermediate values of each function consume several GB of valuable GPU memory. Allocating separate intermediate value memory pools for every function would waste dozens of GB of valuable GPU memory. Therefore, to conserve GPU memory, the Cerium runtime analyzes the call graph of functions in the program and shares this memory pool between non-overlapping function calls.

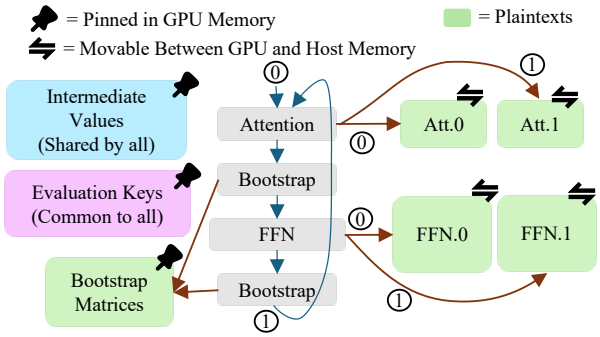


Figure 9. Showing how Cerium manages memory and creates reusable CudaGraphs for a transformer inference

4.8.2 CudaGraphs Generation.

Kernel launch overhead is significant in FHE workloads as even small programs span thousands of kernels. Cerium addresses this using CudaGraphs, an asynchronous programming model which executes a DAG of kernels with a single launch. However, there are several challenges associated with creating CudaGraphs. First, the asynchronous programming model makes manually creating a CudaGraph for thousands of kernels difficult. Second, CudaGraph creation is a significant fraction of the overall execution time and thus must be created offline by the runtime and be reusable across inference queries. Cerium addresses these challenges to reduce kernel launch overheads. It automatically constructs CudaGraphs by leveraging program and memory schedules to infer kernel dependencies (both true and false) and insert edges that enforce happens-before relations, preserving semantics. While such a task is hard to do manually, the compiler and runtime-based approach of Cerium handles it relatively easily. When it comes to the challenges of reusing the graph across program instances and managing graph sizes for large problems, the design of the Cerium's memory layout from Section 4.6 plays an important role. To reuse graphs across programs that use different evalkeys, only the pointer to the evalkey memory pool needs to be updated, thereby avoiding the need to recreate the graph for the new program instance. Similarly, when it comes to dealing with the large graph sizes of BERT-Base and Llama3-8B, the Cerium runtime creates a CudaGraph of each Cerium function and reuses these function-level graphs at every function call. Here, too, Cerium's memory pool design enables CudaGraph reuse by just requiring an update to the plaintext weight pointers.

4.8.3 Memory Pinning and Prefetching.

To support large LLM workloads like encrypted Llama3-8B inference that don't fit in GPU memory, Cerium makes use of unified virtual memory (UVM). However, UVM is slow without guidance, and naively using UVM is detrimental to performance. To address this challenge, the Cerium runtime

uses information from the heterogeneous nature of data lifetimes (Section 4.6) to manage host-to-device memory transfers. Cerium pins frequently-accessed values—such as evalkeys and bootstrap matrices—in GPU memory and shuttles the large weight matrices between the host and GPU memory. The runtime uses information from the compiler and prefetches the weights for each layer before it is run. Figure 9 shows an example of this using a transformer inference. The evalkeys, intermediate values, and bootstrap matrices are all pinned in GPU memory. Before running each layer of the transformer, the runtime updates the pointer to the memory pool and prefetches it from host to GPU.

4.9 Framework Extension

Cerium supports the compilation and GPU execution of CKKS circuits and can implement any application expressible in CKKS. Its compilation pipeline is designed to be extensible and can incorporate complementary techniques that simplify FHE programming, including automatic ciphertext management, such as rescaling and bootstrapping placement [14], and frontends that translate PyTorch or tensor programs into homomorphic circuits [10, 18]. Furthermore, as Cerium's limb IR targets RNS-based modular polynomial arithmetic, it could be extended to support other RNS-RLWE FHE schemes, including BGV [9] and BFV [8, 20].

5 Implementation

We implement the Cerium compiler in ≈25,000 lines of C++ and the runtime in ≈11,000 lines of C++/CUDA. Applications are written in Python using the Cerium DSL. Inter-GPU communication uses NCCL [40].

Compiler. The primary compilation unit is `CeriumFunction`, which is intended to contain coarse-grained functionality like Matrix Multiplication, Non linear functions, Bootstrap, etc and enable modular development of the program. For each `CeriumFunction`, the compiler outputs a shared library of fused kernels, a per-device schedule file detailing kernel arguments and metadata (prime moduli, base conversion, rescale factors, etc.), information on the memory pools (evalkeys, plaintexts, intermediate values), and a call graph of functions. Limb values are formatted as [poolID:offset].

Runtime. During initialization, the runtime loads the `main CeriumFunction` and its callgraph. Then, it loads the information on the memory pool of each called function, allocates memory for them on the GPU, and populates them with values (plaintexts/evalkeys) from files. For the intermediate value pool, it uses the call graph of functions to calculate the maximum size of the pool, and every function is assigned an offset in this pool. This is analogous to a stack frame pointer and enables reusing the intermediate value pool across non-overlapping functions. The runtime then loads the schedule and shared library file for the `CeriumFunctions` in a leaf-to-root order and starts constructing the CudaGraph. The nodes in the CudaGraph correspond to a kernel. The runtime calculates limb value addresses and other metadata and tracks reads and writes by kernels to limb addresses to determine true and false dependencies between kernels and insert edges in the CudaGraph to enforce program semantics. For `CeriumFunctionCalls`, the runtime nests the graph of the CudaGraph of the callee function in the caller CudaGraph.

Execution. After constructing the CudaGraph, the runtime loads the ciphertext inputs to the program, copies them to the GPU, and launches the graph. After execution, the ciphertext outputs are copied back from the GPU to the host. This step can be repeated to execute several instances.

6 Evaluation

We evaluate the following aspects of Cerium:

- Its performance on benchmarks of various sizes
- Comparisons with prior GPU FHE libraries and FHE ASICs
- How long Cerium takes to compile the benchmarks
- A breakdown of Cerium's optimization techniques
- How Cerium's memory management techniques enable and optimize encrypted LLM inference

6.1 Methodology

Benchmarks. We target 128-bit security using ring dimension $N = 64K$, $PQ < 1782$ bits, a ternary main with hamming weight $H = 32K$ and ephemeral secret with $H = 32$. We use 64 28b primes as the RNS basis with $Q < 1446$ bits to implement four benchmarks of increasing sizes.

Bootstrapping [1, 12] refreshes a complex vector ciphertext with a message ratio 2^5 to $l = 18$, consuming 33 levels. It uses a CTS and STC decomposition of 4 and 3, respectively, a 31 degree Chebyshev polynomial and a double angle number of 3 [27] to refresh approximately 20 bits of precision.

ResNet-20 implements a single encrypted classification over a $32 \times 32 \times 3$ CIFAR-10 image. We follow [36] to implement convolutions and approximate ReLU. We achieve an encrypted accuracy of 91.4%, equal to the plaintext accuracy.

BERT implements an encrypted BERT-Base [17] inference on a 128 token input. We use [50] to approximate nonlinearities. BERT-Base contains a total of 997 bootstraps. For high accuracy, we implement softmax with max subtraction. We achieve an encrypted inference accuracy of 69.3%, on the GLUE RTE dataset [6, 48], matching the plaintext accuracy.

Llama3-8B implements the prefill stage of an encrypted inference over a 128 token prompt. We report the time to first token (TTFT). Llama3-8B contains a total of 5,890 bootstraps. We do not use any modifications that require retraining the model. We validate the encrypted inference using zero-shot MCQs as the limited token budget prevents us from running standard LLM benchmarks.

Configurations. We evaluate Cerium across modern datacenter systems equipped with SXM GPUs: DGX A100 (80GB), DGX H100, and DGX B200, with 1.1, 2.0, and 2.0 TB of host

Benchmark	GPU	Cerium Execution Time				Related Work	Unencrypted	
		1×	2×	4×	8×		CPU	1× GPU
Bootstrap	B200	14.5 ms	9.9 ms	8.2 ms	7.5 ms	C: 21 ms	-	-
	H100	19.5 ms	13.7 ms	10.6 ms	9.7 ms	C: 23 ms		
	A100	34.2 ms	22.5 ms	17.9 ms	16.5 ms	C: 40 ms		
ResNet-20	B200	456 ms	340 ms	310 ms	298 ms	-	5 ms	2 ms
	H100	620 ms	464 ms	408 ms	386 ms	-		3 ms
	A100	1050 ms	774 ms	676 ms	638 ms	C: 1320 ms		4 ms
BERT-Base	B200	28.3 s	19.6 s	10.8 s	8.8 s	-	0.5 s	0.005 s
	H100	36.1 s	23.91 s	13.3 s	10.6 s	-		0.007 s
	A100	66.1 s	44.3 s	24.6 s	19.6 s	T: 602 s		0.009 s
Llama3-8B	B200	241 s	147 s	71 s	43 s	-	1.5 s	0.02 s
	H100	370 s	206 s	92 s	62 s	-		0.03 s
	A100	555 s	351 s	155 s	111 s	N: 13,271 s		0.04 s

C=Cheddar[1] [15](1×), T=THOR [38](1×A100), N=Nexus [50](4×A100)

Table 1. Benchmark Execution Times

memory, respectively. We parallelize workloads across 1, 2, 4 and 8 GPUs. The encoded model weights and evalkeys are computed offline and reside on the server.

6.2 Comparisons with Prior GPU FHE Work

Table 1 reports benchmark execution time across datacenter GPUs. Cerium achieves an average 1.21× speedup over Cheddar on a single A100 GPU, demonstrating that its automated kernel fusion, scheduling, and memory orchestration can surpass expert hand-tuned implementations. With multi-GPU parallelization, Cerium further improves performance to 2.25× over Cheddar. For BERT-Base, Cerium delivers a 9.12× speedup, reducing inference latency from THOR's 602.2 s to 66 s on 1×A100. Beyond optimized kernel generation, these improvements come from Cerium's memory management techniques (Sections 4.5, 4.6 and 4.8), which allow the model to fit within GPU memory and run end-to-end. Multi-GPU scaling provides an additional 3.36× speedup, reducing BERT-Base inference to 19.6 s. For Llama3-8B, prior work does not report end-to-end encrypted inference. The closest comparison, Nexus [50], estimates performance by aggregating layer runtimes, without accounting for memory overheads, for an 8 token inference. When normalized to 128 tokens, Cerium is 85.6× faster.

6.3 Comparison with FHE ASICs

Figure 10 compares Cerium with the FHE ASIC designs Cinnamon [28], ARK [32], and CraterLake [44], with execution times normalized to Cinnamon-8. The ASIC baselines are derived from simulation studies rather than silicon measurements; therefore, the comparison reflects projected ASIC performance.

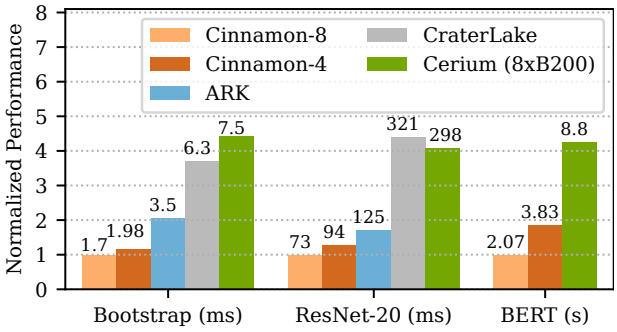


Figure 10. Comparing Cerium to FHE ASICs [2]

Using 8× B200 GPUs, Cerium achieves average speedups of 1.06×, 2.33×, 3.03×, and 4.40× over CraterLake, ARK, Cinnamon-4, and Cinnamon-8, respectively. These results show that a highly optimized software stack can make commercial GPU hardware competitive with specialized FHE accelerators. To the best of our knowledge, Cerium is the first system to demonstrate sub-10 ms bootstrapping on commercial hardware, achieving 7.5 ms, as well as encrypted BERT-Base inference in under 10 s, achieving 8.8 s.

6.4 Compile Time and CudaGraph Creation Time

Table 2 reports the compilation time of the Cerium framework and the corresponding CudaGraph creation time across our benchmarks. Cerium compiles small programs such as Bootstrap and ResNet-20 within seconds (5.5 s and 13.9 s, respectively), and scales efficiently to large encrypted workloads, compiling BERT-Base and Llama3-8B in 2 min 47 s and 11 min 29 s, respectively. The runtime's CudaGraph creation

[1]For Cheddar, we ran the bootstrapping benchmark as it is open sourced. Cheddar's ResNet-20 implementation has not been released at the time of writing and the ResNet-20 results are taken from the paper.

[2]CraterLake and ARK results are omitted for BERT-Base because the published papers do not evaluate this benchmark and their implementations are not publicly available.

Benchmark	Cerium Compile Time	CudaGraph Creation Time
Bootstrap	5.5 s	350 ms
ResNet-20	13.9 s	1.75 s
BERT-Base	2 min 47 s	4.75 s
Llama3-8B	11 min 29 s	24.45 s

Table 2. Compilation and CudaGraph Creation Times

incurs minimal overhead, completing in under 30 s even for Llama3-8B. In contrast, manually implementing and optimizing these kernels and graphs would require several hours or days of expert GPU and FHE programming effort. As CudaGraph creation is a significant portion of overall runtime, it must be amortizable across several inferences. Cerium's memory pool architecture enables the same CudaGraph to be reused for several successive inferences at the cost of a few pointer updates to the relevant pools (Section 4.8.2). These results demonstrate that Cerium can rapidly generate optimized, end-to-end GPU implementations from circuit-level FHE descriptions, significantly accelerating development cycles and enabling non-expert programmers to efficiently deploy complex encrypted workloads.

6.5 Cerium Techniques Evaluation

In this section, we evaluate the techniques proposed in this paper. All experiments are performed on the 1×B200 GPU configuration, except for the multi-GPU evaluation.

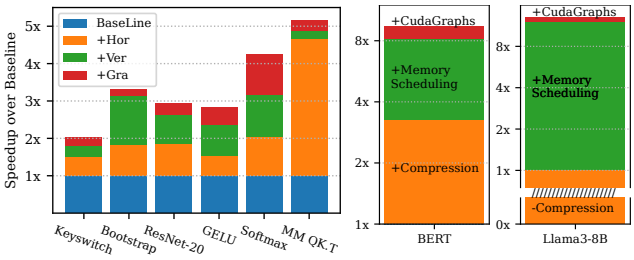


Figure 11. Cerium Techniques: (a) Kernel Optimization; Memory Optimizations for (b) BERT-Base, (c) Llama3-8B

6.5.1 Horizontal and Vertical Fusion. To assess Cerium's horizontal and vertical fusion passes, we evaluate six workloads: keyswitching ($l = 48, d = 4$), bootstrapping, ResNet-20, GELU, softmax and Ciphertext-Ciphertext Matmul (MM QK.T). GELU, softmax and MM QK.T are taken from the BERT-Base benchmark; for softmax, we report performance excluding internal bootstrapping time. The baseline is an unfused implementation where each polynomial operation is lowered to a separate kernel. For bootstrapping and ResNet-20, horizontal fusion provides speedups of 1.98× and 1.84×, respectively. Vertical fusion adds another 1.68× and 1.43×, resulting in overall improvements of 3.32× for bootstrapping and 2.63× for ResNet-20. For GELU, the unfused implementation takes 84.98 ms. Horizontal fusion reduces the runtime to 54.87 ms (1.54×), and vertical fusion reduces it further to 36.13 ms (1.51× over horizontal). For softmax, the unfused version takes 235.4 ms. Horizontal fusion lowers this to 115.11 ms (2.04×), and vertical fusion further reduces the time to 74.09 ms (1.55× over horizontal). MM QK.T (ciphertext multiplication between 128x128 matrices) shows a 4.8× speedup (195 ms to 40.8 ms) from Horizontal fusion as it is a very wide benchmark with a small depth. In all cases, the speedup is significantly larger than the speedup of a single keyswitch, demonstrating the importance of fusion across large sections of the program and the versatility of Cerium's compiler-driven approach.

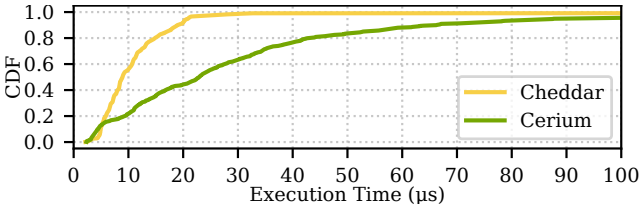


Figure 12. CDF for Bootstrapping Kernels

6.5.2 Automatic Fusion. To evaluate the benefits of Cerium's automated kernel fusion, we compare its compiler-generated kernels with Cheddar's [15] manually fused kernels. Cerium reduces the number of kernels required for bootstrapping from 1490 in Cheddar to 472, a 3.1× reduction. Figure 12 presents the CDF of kernel execution times for both systems on bootstrapping. In Cheddar, 56.1% of kernels execute in under 10 µs and 97.5% complete within 25 µs, indicating a large number of short-running kernels. This reflects the limitations of manual fusion, where identifying opportunities across larger program regions is difficult. For instance, Cheddar doesn't support fusion across the two chebyshev polynomials. In contrast, Cerium discovers fusion opportunities across the entire program to generate larger kernels, with 21.7% executing under 10 µs and 55.4% under 25 µs. Consequently, Cerium achieves higher average SM active cycles of 94.1%, compared to Cheddar's 72.1%, resulting in a 22% speedup.

6.5.3 CudaGraphs. To mitigate kernel launch overheads, the Cerium runtime uses CudaGraphs to specify the computation as an asynchronous graph of kernels that can be executed with a single low-overhead launch. CudaGraphs yield consistent performance gains across all benchmarks: a 7% speedup for bootstrapping, a 12% speedup for ResNet-20, a 14% speedup for BERT-Base, and a 10% speedup for Llama3-8B, resulting in an average improvement of 11%.

6.5.4 Plaintext Compressed Encoding. Packing plaintext vectors with power-of-two periods enables the Cerium

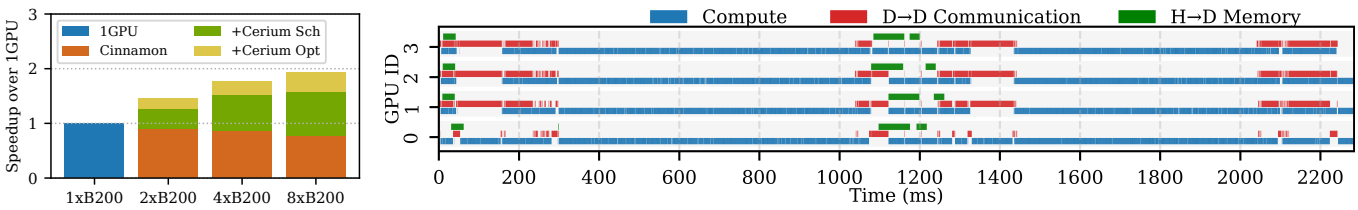


Figure 13. (a) Cerium's Multi-GPU Optimizations; (b) Llama3-8B Transformer Block Execution Timeline

compiler to compress them without changing the computation. This optimization has two major benefits. First, it dramatically shrinks the memory footprint of encoded model weights, making previously intractable models feasible to run under FHE. For BERT-Base, this technique achieves a 96× reduction, compressing 1.5 TB of encoded weights down to 16.6 GB — small enough to fit entirely in GPU memory. Second, compression reduces pressure on memory bandwidth. Collectively, compression improves performance by 3.25× from 266.4 s to 81.68 s. The effect is even more pronounced for larger models. For Llama3-8B, plaintext compression reduces the encoded weight footprint by 119× from 112 TB to 982 GB. Without this reduction, the model would exceed GPU and host memory capacity by two orders of magnitude, making end-to-end FHE inference entirely infeasible. These results highlight how Cerium's compiler exploits structural properties in plaintext encodings to unlock substantial practical scalability in FHE workloads.

6.5.5 Memory Schedule. To avoid the overhead of repeatedly allocating and freeing intermediate ciphertexts, Cerium constructs a kernel and memory schedule that captures data dependencies across kernels. The runtime uses this schedule to reuse memory locations and eliminate online allocation. We evaluate this approach against a baseline that allocates and frees memory for every intermediate value. Online memory management slows bootstrapping from 15.1 ms to 35.57 ms, a 2.3× slowdown. For ResNet-20, the slowdown is 2.36× (509 ms to 1206 ms). For BERT, reusing memory locations leads to a further speedup of 2.51× from 81.68 to 32.55 seconds. These results demonstrate the importance of Cerium's compiler and runtime directed memory reuse, which eliminates the high cost of online memory operations.

Llama3-8B has a working set size that far exceeds the capacity of GPUs. Cerium addresses this challenge by using the memory layout, pinning, and prefetching techniques described in Sections 4.6 and 4.8 to coordinate data movement between host and GPU. When evaluated against a baseline that uses UVM for all allocations without prefetching, Cerium's optimizations yield a 12.1× speedup for Llama3-8B, reducing execution time from 3334 to 275 seconds and highlighting the importance of a memory management system.

6.5.6 Multi GPU Parallelization and Scheduling. We evaluate multi-GPU parallelized bootstrapping using B200 GPUs. Our baselines are a single-GPU baseline and a basic multi-GPU implementation using Cinnamon's [28] parallel keyswitching algorithm in a single stream. On average, the baseline implementation performs 1.2× slower than a single GPU's 14.5 ms. However, when enabling Cerium's multi-GPU scheduling techniques to reorder and overlap compute and communication operations, performance improves on average to 1.45× the single GPU case. Furthermore, using Cerium's new communication minimization passes results in a 44% reduction in bytes communicated over Cinnamon and improves performance by an additional 1.18×.

Figure 13(b) shows the timeline of a Llama3-8B transformer block across four B200 GPUs. Cerium overlaps computation with device-to-device communication and host-to-device transfers, hiding 99% of host-to-device transfer latency and 58% of device-to-device communication latency.

Our evaluation demonstrates that in order to accelerate FHE over multiple GPUs and close the performance gap with FHE ASICs, the algorithmic techniques from Cinnamon alone are not enough and they require a co-designed compiler and runtime system to deliver meaningful speedups.

7 Other Related Work

CROSS [47] targets FHE primitive acceleration on TPUs, but does not run end-to-end applications. Cerium in contrast, runs end-to-end applications up to Llama3-8B. GME [45] introduces 186.2 mm^2 of additional logic to an AMD MI100 to support FHE yet trails ASICs by more than 10×. In contrast, Cerium approaches within 3× of FHE ASIC performance without any hardware modifications, relying solely on software optimizations. BOLT [42] combines HE and MPC for transformer inference, but its online MPC communication costs dominate: even on a LAN, 128-token BERT-Base inference requires 91 s. Cerium shows that the same workload can now run fully in FHE on GPUs in 8.8 s, avoiding the networking bottleneck of MPC. Concurrent efforts [38, 43, 50] explore FHE transformer inference, but progress has been constrained by the absence of a general, high-performance GPU framework. By automatically generating optimized GPU implementations, Cerium aims to unlock faster research and deployment of FHE-based applications.

Gouert et al. [25] target Boolean-gate FHE, showing multi-GPU scaling for small non-ML benchmarks. MAGE [35] exploits the static nature of memory access patterns of encrypted computation to optimize disk-memory transfers. Finally, we remark that as FHE ASICs [28, 32, 44] are statically scheduled, they cannot support workloads like Llama3-8B that exceed accelerator memory and require host-to-accelerator memory orchestration, highlighting a major scalability limitation.

8 Potential Use Cases for FHE

FHE is particularly compelling when the value of confidentiality exceeds the performance overheads. This holds, for instance, when organizations hold valuable data that cannot be outsourced in plaintext due to confidentiality, intellectual property, or regulatory concerns. Cerium demonstrates FHE-based encrypted inference for transformer models at the scale of BERT-Base and Llama3-8B, with processing times on the order of tens of seconds. While this latency is unsuitable for most interactive services, it is practical for offline analysis, asynchronous review, and batch processing, where protecting sensitive data can matter more than quick response times. We illustrate a few such use cases.

Consider a hospital running encrypted inference on a genomic-risk or disease-prediction model. The result may inform a clinical decision or prioritize further review, both of which can tolerate several seconds of latency. However, inputs such as genetic data, medical images, and patient records may be too sensitive to disclose to an external model provider [22, 23, 34]. Similarly, a law firm could use encrypted inference during legal discovery to identify relevant documents within a client's internal corpus without needing to view the entire corpus [4]. Here, tens of seconds of processing time is acceptable relative to a discovery process that typically spans weeks.

Similar situations arise in financial analysis and software security. A financial institution could run an anti-money-laundering or credit-risk model over encrypted customer transaction data, where a delay of dozens of seconds is minor compared with the consequences of exposing sensitive financial records [39, 41]. Likewise, an organization could submit encrypted proprietary source code to an AI-assisted vulnerability-detection service, enabling the service to identify potential vulnerabilities in the code without revealing it to the service provider [33]. As vulnerability analysis can be performed offline, several seconds of latency is acceptable.

9 Future Work

Cerium demonstrates encrypted transformer inference at the scale of BERT-Base and Llama3-8B, but several challenges remain to enable broader deployment. One important direction is efficient encrypted KV-cache management for autoregressive decoding, including the design of suitable packing strategies and the management of the capacity and data-movement costs associated with subsequent-token generation. Another is studying the tradeoff between model accuracy and the computational cost of polynomial approximations for nonlinear operations, including robustness to adversarial and out-of-distribution inputs. Cerium provides a framework for evaluating the performance of alternative packing strategies and polynomial approximations alongside their effects on model accuracy.

Future work could also improve energy efficiency through hardware- and compiler-level techniques that reduce the cost of encrypted arithmetic, communication, and memory movement. Finally, continuous batching of encrypted requests could improve GPU utilization and system throughput, although it requires careful management of the substantial memory footprint of concurrent FHE computations. These directions could build on Cerium's limb-level IR, kernel-fusion passes, and memory-pool design.

10 Conclusion

FHE usage has been constrained by the assumption that commodity hardware is insufficient and custom silicon is required. By building a highly optimized software stack for GPUs, Cerium is able to match the performance of FHE ASICs – showing that the performance gap between commodity GPUs and custom silicon can be bridged entirely in software and without building custom hardware. The Cerium framework automates the creation of optimized multi-GPU FHE programs using compiler techniques for kernel generation, optimization, and scheduling. Cerium introduces techniques that exploit algebraic structure to reduce the memory capacity requirements of encrypted inference by more than 100×, minimize the memory overhead of FHE programs, and enable running terabyte-scale encrypted inference workloads for the first time.

Acknowledgments

We would like to thank the anonymous reviewers and shepherd for feedback on the paper; Edward Chen for comments, feedback and help drafting the manuscript. This research was funded by NSF Grants CNS-2238671 and CCF 2217016, an Amazon Web Services Grant, and a CMU Cylab Grant.

References

[1] 2023. Lattigo v5. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.

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

[3] Carlos Agulló-Domingo, Óscar Vera-López, Seyda Guzelhan, Lohit Daksha, Aymane El Jerari, Kaustubh Shivdikar, Rashmi Agrawal, David

Kaeli, Ajay Joshi, and José L Abellán. 2025. FIDESlib: A Fully-Fledged Open-Source FHE Library for Efficient CKKS on GPUs. 3 pages.

[4] Adi Akavia, Craig Gentry, Shai Halevi, and Max Leibovich. 2019. Setup-Free Secure Search on Encrypted Data: Faster and Post-Processing Free. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 87–107. doi:10.2478/popets-2019-0038

[5] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Cryptology ePrint Archive, Paper 2016/510. https://eprint.iacr.org/2016/510

[6] Luisa Bentivogli, Ido Dagan, Hoa Trang Dang, Danilo Giampiccolo, and Bernardo Magnini. 2009. The Fifth PASCAL Recognizing Textual Entailment Challenge.

[7] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2021. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I* (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 587–617. doi:10.1007/978-3-030-77870-5_21

[8] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. 19 pages. doi:10.1007/978-3-642-32009-5_50

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

[10] Edward Chen, Fraser Brown, and Wenting Zheng. 2025. Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption. Cryptology ePrint Archive, Paper 2025/1319. https://eprint.iacr.org/2025/1319

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2018/153. https://eprint.iacr.org/2018/153

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2018/153. https://eprint.iacr.org/2018/153

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.

[14] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. 2024. DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 6993–7010. https://www.usenix.org/conference/usenixsecurity24/presentation/cheon

[15] Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. 2024. Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures. 15 pages, 8 figures; accepted at ASPLOS 2026. arXiv:2407.13055 [cs.CR] doi:10.48550/arXiv.2407.13055

[16] DESILO. 2023. Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy. https://github.com/Desilo/liberate-fhe.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805

[18] Austin Ebel, Karthik Garimella, and Brandon Reagen. 2025. Orion: A Fully Homomorphic Encryption Framework for Deep Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 734–749. doi:10.1145/3676641.3716008

[19] Guang Fan, Mingzhe Zhang, Fangyu Zheng, Shengyu Fan, Tian Zhou, Xianglong Deng, Wenxu Tang, Liang Kong, Yixuan Song, and Shoumeng Yan. 2025. WarpDrive: GPU-Based Fully Homomorphic Encryption Acceleration Leveraging Tensor and CUDA Cores. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1187–1200. doi:10.1109/HPCA61900.2025.00091

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144

[21] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 922–934. doi:10.1109/HPCA56546.2023.10071017

[22] David Froelicher, Juan R. Troncoso-Pastoriza, Jean Louis Raisaro, Michel A. Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. 2021. Truly Privacy-Preserving Federated Analytics for Precision Medicine with Multiparty Homomorphic Encryption. *Nature Communications* 12 (2021). doi:10.1038/s41467-021-25972-y

[23] Ravit Geva, Alexander Gusev, Yuriy Polyakov, Lior Liram, Oded Rosolio, Andreea Alexandru, Nicholas Genise, Marcelo Blatt, Zohar Duchin, Barliz Waissengrin, Dan Mirelman, Felix Bukstein, Deborah T. Blumenthal, Ido Wolf, Sharon Pelles-Avraham, Tali Schaffer, Lee A. Lavi, Daniele Micciancio, Vinod Vaikuntanathan, Ahmad Al Badawi, and Shafi Goldwasser. 2023. Collaborative privacy-preserving analysis of oncological data using multiparty homomorphic encryption. *Proceedings of the National Academy of Sciences* 120, 33 (2023), e2304415120. doi:10.1073/pnas.2304415120

[24] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*. PMLR, New York, New York, USA, 201–210. https://proceedings.mlr.press/v48/gilad-bachrach16.html

[25] Charles Gouert, Vinu Joseph, Steven Dalton, Cedric Augonnet, Michael Garland, and Nektarios Georgios Tsoutsos. 2025. Hardware-accelerated encrypted execution of general-purpose applications. *Proceedings on Privacy Enhancing Technologies* (2025).

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

[27] Kyoohyung Han and Dohyeong Ki. 2019. Better Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2019/688. https://eprint.iacr.org/2019/688

[28] Siddharth Jayashankar, Edward Chen, Tom Tang, Wenting Zheng, and Dimitrios Skarlatos. 2025. Cinnamon: A Framework for Scale-Out Encrypted AI. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 133–150. doi:10.1145/3669940.3707260

[29] Dian Jiao, Xianglong Deng, Zhiwei Wang, Shengyu Fan, Yi Chen, Dan Meng, Rui Hou, and Mingzhe Zhang. 2025. Neo: Towards Efficient Fully Homomorphic Encryption Acceleration using Tensor Core. In *Proceedings of the 52nd Annual International Symposium on Computer*

Architecture (ISCA '25). Association for Computing Machinery, New York, NY, USA, 107–121. doi:10.1145/3695053.3731408

[30] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. Cryptology ePrint Archive, Paper 2021/508. https://eprint.iacr.org/2021/508

[31] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. doi:10.1145/3579371.3589053

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

[33] Seungho Kim, Seonhye Park, Jihun Kim, Eunsoo Kim, Sanghak Oh, Hyunmin Choi, and Hyoungshick Kim. 2026. PP-Vul: Privacy-Preserving Vulnerability Detection Using Homomorphic Encryption. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. 1415–1430. doi:10.1145/3779208.3785415

[34] Elizabeth Knight, Jiaqi Li, Matthew Jensen, Israel Yolou, Can Kockan, and Mark Gerstein. 2026. Homomorphic Encryption Enables Privacy Preserving Polygenic Risk Scores. *Cell Reports Methods* 6 (2026). doi:10.1016/j.crmeth.2025.101271

[35] Sam Kumar, David E. Culler, and Raluca Ada Popa. 2021. MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation. 367–385 pages. https://www.usenix.org/conference/osdi21/presentation/kumar

[36] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2021. Privacy-Preserving Machine Learning with Fully Homomorphic Encryption for Deep Neural Network. *CoRR* abs/2106.07229 (2021). arXiv:2106.07229 https://arxiv.org/abs/2106.07229

[37] Guillermo Lloret-Talavera, Marc Jorda, Harald Servat, Fabian Boemer, Chetan Chauhan, Shigeki Tomishima, Nilesh N. Shah, and Antonio J. Pena. 2022. Enabling Homomorphically Encrypted Inference for Large DNN Models. *IEEE Trans. Comput.* 71, 5 (May 2022), 1145–1155. doi:10.1109/tc.2021.3076123

[38] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. 2024. THOR: Secure Transformer Inference with Homomorphic Encryption. Cryptology ePrint Archive, Paper 2024/1881. doi:10.1145/3719027.3765150

[39] National Institute of Standards and Technology. 2026. Privacy-Enhancing Cryptography: Fully-Homomorphic Encryption (FHE). Accessed: 2026-08-26. https://csrc.nist.gov/projects/pec/fhe

[40] NVIDIA. 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[41] OECD. 2025. *Sharing Trustworthy AI Models with Privacy-Enhancing Technologies*. Technical Report 38. OECD Publishing. doi:10.1787/a266160b-en

[42] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. 2023. BOLT: Privacy-Preserving, Accurate and Efficient Inference for Transformers. Cryptology ePrint Archive, Paper 2023/1893. https://eprint.iacr.org/2023/1893

[43] Donghwan Rho, Taeseong Kim, Minje Park, Jung Woo Kim, Hyunsik Chae, Ernest K. Ryu, and Jung Hee Cheon. 2025. Encryption-Friendly LLM Architecture. arXiv:2410.02486 [cs.CR] https://arxiv.org/abs/2410.02486

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

[45] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. 2023. GME: GPU-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the International Symposium on Microarchitecture*. 670–684.

[46] Taiwan Semiconductor Manufacturing Company Limited. 2024. CoWoS®: Chip on Wafer on Substrate Technology. https://3dfabric.tsmc.com/english/dedicatedFoundry/technology/cowos.htm.

[47] Jianming Tong, Tianhao Huang, Leo de Castro, Anirudh Itagi, Jingtian Dang, Anupam Golder, Asra Ali, Jevin Jiang, Arvind, G. Edward Suh, and Tushar Krishna. 2025. Leveraging ASIC AI Chips for Homomorphic Encryption. arXiv:2501.07047 [cs.CR] https://arxiv.org/abs/2501.07047

[48] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In the Proceedings of ICLR..

[49] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. 2023. Phantom: A CUDA-Accelerated Word-Wise Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2023/049. https://eprint.iacr.org/2023/049. doi:10.1109/TDSC.2024.3363900

[50] Jiawen Zhang, Jian Liu, Xinpeng Yang, Yinghao Wang, Kejia Chen, Xiaoyang Hou, Kui Ren, and Xiaohu Yang. 2024. Secure Transformer Inference Made Non-interactive. Cryptology ePrint Archive, Paper 2024/136. https://eprint.iacr.org/2024/136

Note: This appendix contains supplementary material that has not been peer-reviewed.

A Plaintext Compressed Encoding

A.1 Proof of Periodic Compression

Let $V \in \mathbb{C}^M$ be T-periodic, where $T < M$ are powers of 2. Let $y(X) \in \mathbb{C}[X]/(X^N + 1)$ denote its CKKS encoding, for $N = 2M$. We want to show that the periodicity of V induces: 1) coefficient sparsity in y, with support restricted to indices spaced by $\frac{N}{2T}$, and 2) periodic structure in the NTT domain, with evaluation values repeating every $\frac{N}{2T}$ slots.

Lemma 2. *Let X be a permutation of V with index map of $k(i) := \frac{(5^i \bmod 4M)-1}{4}, 0 \le i < M$. If V is periodic with period T, then X is also periodic with period T.*

Proof. First, $k(i)$ is a valid permutation as $5^i \pmod{4M}$ generates M unique odd integers of the form $4k+1$ for $k \in [0, M-1)$. Given index $i_l \in [0, M)$, let $k_l = k(i_l)$. WLOG, let $i_2 = i_1 + \lambda T$, for $\lambda \in \mathbb{Z}$. Then, $k_2 - k_1 = (5^{i_2} - 5^{i_1})/4 = 5^{i_1}((1+4)^{\lambda T} - 1)/4$. Applying the Binomial Theorem, we get $k_2 - k_1 = 5^{i_1}\left(\left(\sum_{j=0}^{\lambda T}\binom{\lambda T}{j}4^j\right) - 1\right)/4 = 5^{i_1} \cdot 4T\left(\lambda + \frac{\lambda(\lambda T-1)}{2} \cdot 4 + \ldots\right)/4$. Let the integer term in the blue parentheses be m. Then $k_2 - k_1 = 5^{i_1} \cdot Tm \equiv 0 \pmod{T}$. Since the permutation is bijective, different equivalence classes in X cannot fall into the same equivalence class in V. □

Let Y be the conjugate part extension vector of X, formally defined as: $Y[2k] = X[k] \quad \text{and} \quad Y[2(M-k)-1] = \overline{X[k]}$. By definition, Y is of size $N = 2M$ with a period $2T$.

Lemma 3 (Sparsity of IFFT on Y)**.** *Let N and $2T$ be powers of two such that $2T \mid N$. Let ω be a primitive $2N$-th root of unity. Let y be the IFFT of Y, where $y[\hat{j}] = \frac{1}{N}\sum_{j=0}^{N-1} Y[j]\,\omega^{-\hat{j}(2j+1)}$. If Y is periodic with period $2T$, then y is a $2T$-sparse polynomial.*

Proof. We can rewrite the IFFT summation by breaking it into two nested summations: $y[\hat{j}] = \frac{1}{N}\sum_{t=0}^{2T-1}\sum_{s=0}^{(N/2T)-1} Y[t + s(2T)]\omega^{-\hat{j}(2(t+s(2T))+1)}$. Since Y is periodic with period $2T$, $Y[t] = Y[t + s(2T)]$. Thus:

$$y[\hat{j}] = \frac{1}{N}\sum_{t=0}^{2T-1} Y[t]\omega^{-\hat{j}(2t+1)}\left(\sum_{s=0}^{(N/2T)-1}(\omega^{-4T\hat{j}})^s\right)$$

Using the orthogonality properties of the roots of unity, we can conclude that the inner summation (in blue) is non-zero only when $\hat{j} \equiv 0 (\bmod\ N/2T)$; and thus y is a $2T$-sparse polynomial. □

Moreover, for a sparse vector y, scaling, rounding, and integer modulus preserve sparsity.

Lemma 4 (Periodicity of NTT on y)**.** *Let N and $2T$ be powers of two such that $2T \mid N$. For polynomial y of degree N, let V_{ntt}, also of size N, be the set of polynomial evaluations at the roots of unity ω. If y is structured $2T$-sparse, i.e., $y[j] = 0$ unless $j \equiv 0 \bmod \frac{N}{2T}$, then V_{ntt} is periodic with period $2T$.*

Proof. Use the property that $\omega^{(2j+1)N/2T}$ is $2T$-periodic. □

Lemma 5 (Contiguity of Bit Reversal on Periodic Vectors)**.** *Let N and $2T$ be powers of 2, and V_{ntt} be a periodic vector of size N and $2T$. Then a bit reversal transformation on V_{ntt} results in a contiguous vector.*

Proof. By periodicity, $i_1 \equiv i_2 \pmod{2T} \implies V_{ntt}[i_1] = V_{ntt}[i_2]$. Also, since N and $2T$ are powers of 2, lower $log_2(2T)$ bits of i_1 and i_2 are equal. Thus after bit reversal, the upper $log_2(2T)$ bits of i_1 and i_2 will be the same, forming a contiguous vector. □

Therefore, given a periodic plaintext vector V, Lemma 2 shows the preservation of periodicity after permutation. Then applying Lemmas 3, 4 show the preservation of periodicity (albeit doubled) throughout the CKKS encoding. Finally, Lemma 5 shows that bit reversal on the periodic NTT representation creates a contiguous vector, which sets up the vector for compression.

A.2 Homomorphic Operations over Compressed Plaintext Encodings

Note that the NTT representation is used for efficient ciphertext multiplication. Figure 14 shows how Cerium implements ciphertext-plaintext multiplication when the plaintext limb vector is compressed. Homorphic addition can be implemented similarly.

```
# Inputs: Ciphertext C = (C_0, C_1) in RNS-NTT form
#  Plaintext P_com with period T in Compressed Form
def Ct_CompPt_Mul(C, P_com, T):
  for j,i in (C.num_limbs × N):
     D_0[j][i] = C_0[j][i] * P_com[j][i >> log2(M/T)]
     D_1[j][i] = C_1[j][i] * P_com[j][i >> log2(M/T)]
  return (D_0, D_1)
```

Figure 14. Ciphertext - Compressed Plaintext Multiplication

The algorithm in Figure 14 can be extended to matrix multiplication when plaintext matrices are carefully packed into vectors with powers of two periods. Figure 15 describes ciphertext-compressed plaintext multiplication with the help of an example where two tensors A and B of size $(2, n = 8, n = 8)$ are multiplied using $M = 2n^2 = 128$ slots. The operation performs batched multiplication of the two $n \times n$ tiles. Here, A is the ciphertext and B is the plaintext. We pack both tiles of A interleaved row-wise as shown in A_p. B is packed in a periodic diagonal form. Plaintext B_i contains periodic packing of the i^{th} lower diagonal of both tiles of B with no wrap around. Negative values of i correspond to upper diagonals. We zero pad lower diagonals at the end and upper diagonals at the start. The matmul can be written as $\sum_{i=-n}^{n}(A_p << i) \cdot B_i$ and the result is row-packed, similar to A. The rotations required can be optimized using BSGS to $2\sqrt{2n} = 8$ rotations. This method requires $2n - 1 = 15$

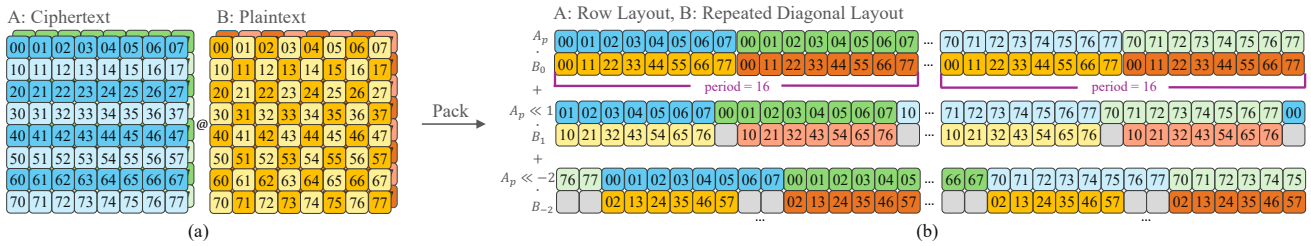


Figure 15. Matrix Multiplication with Periodic Plaintexts

plaintext vectors, each of which is periodic with $T = 2n = 16$. Thus, they can be compressed by a factor $M/T = n = 8$, resulting in an effective plaintext count of $2 - 1/n$. Larger matrices can be padded and broken up into similar tiles to implement matrix multiplication.

A.3 Generality of Power of 2 Periodic Packings

As most tensor operations in FHE and AI are performed over tensors sizes that are multiples of powers of 2, Cerium's plaintext compression is generalizable to most workloads. In case tensors are not powers of 2, they can be padded to a multiple of a power of 2 to implement the compression.