2026 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Leveraging ASIC AI Chips for Homomorphic Encryption

Jianming Tong*†, Tianhao Huang†, Jingtian Dang*, Leo de Castro†, Anirudh Itagi*, Anupam Golder*, Asra Ali‡
Jeremy Kun‡, Jevin Jiang‡, Arvind†¶ G. Edward Suh§ , Tushar Krishna*
*Georgia Institute of Technology †Massachusetts Institute of Technology ‡Google §NVIDIA/Cornell University
jianming.tong@gatech.edu, tianhaoh@mit.edu, {asraa, jkun, jevinjiang}@google.com, suh@ece.cornell.edu, tushar@ece.gatech.edu

***Abstract*— Homomorphic Encryption (HE) provides strong data privacy for cloud services but at the cost of prohibitive computational overhead. While GPUs have emerged as a practical platform for accelerating HE, there remains an order-of-magnitude energy-efficiency gap compared to specialized (but expensive) HE ASICs. This paper explores an alternate direction: leveraging existing AI accelerators, like Google's TPUs with coarse-grained compute and memory architectures, to offer a path toward ASIC-level energy efficiency for HE.**

However, this architectural paradigm creates a fundamental mismatch with SoTA HE algorithms designed for GPUs. These algorithms rely heavily on: (1) high-precision (32-bit) integer arithmetic to now run on a TPU's low-throughput vector unit, leaving its high-throughput low-precision (8-bit) matrix engine (MXU) idle, and (2) fine-grained data permutations that are inefficient on the TPU's coarse-grained memory subsystem. Consequently, porting GPU-optimized HE libraries to TPUs results in severe resource under-utilization and performance degradation.

To tackle above challenges, we introduce CROSS, a compiler framework that systematically transforms HE workloads to align with the TPU's architecture. CROSS makes two key contributions: (1) *Basis-Aligned Transformation (BAT)*, a novel technique that converts high-precision modular arithmetic into dense, low-precision (INT8) matrix multiplications, unlocking and improving the utilization of TPU's MXU for HE, and (2) *Memory-Aligned Transformation (MAT)*, which eliminates costly runtime data reordering by embedding reordering into compute kernels through offline parameter transformation.

Our evaluation on a real single-host Google TPU v6e refreshes the SoTA Number Theoretic Transform (NTT) throughput record with up-to 1.43× *throughput improvement* over WarpDrive on a NVIDIA A100. Furthermore, CROSS achieves 451×, 7.81×, 1.83×, 1.31×, 1.86×, and 1.15× higher throughput per watt than OpenFHE, WarpDrive, FIDESlib, FAB, HEAP, and Cheddar, respectively, establishing AI ASIC as the SotA efficient platform for HE operators. Code: https://github.com/EfficientPPML/CROSS.

I. Introduction

Artificial Intelligence (AI) is driving a new industrial revolution, transforming how we create, exchange, and safeguard information. From chatbots [29], [51], generative model [57] to AI coders [47], human workflows are increasingly translated into digital tokens—a process that **makes the world effectively tokenized**. This revolution necessitates AI systems that learn and operate on sensitive, individual user data, making robust privacy preservation not merely a feature, but a foundational requirement. Homomorphic Encryption (HE) offers a powerful solution by enabling direct computation on encrypted data, as shown in Fig. 1. But its practical adoption is hindered by extreme performance overheads, with data and computation costs inflating by orders of magnitude, leading to 1000× slowdown on a multi-core CPU [8], [50], [55], [56].

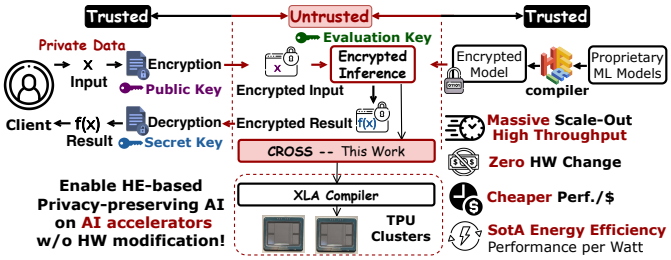


Figure 1: CROSS enables direct computation on encrypted data to enable privacy-preserving model serving on AI ASICs.

Dedicated ASICs [3], [31], [38], [39], [41], [52], [56], [58], [59], [64], [69] have been proposed to tackle this challenge, but their high design and fabrication costs present a significant barrier to widespread deployment. Consequently, commodity hardware like GPUs [5], [6], [11], [17], [21], [24], [26], [46], [63], [65], [66], [68], [74] and FPGAs [4], [53], [54] have become the de-facto platforms for high-performance HE, achieving SoTA throughput and energy efficiency. While still approximately 33× less energy-efficient than a well-designed HE ASIC (Tab. VIII), the lack of commercial HE chips and the widespread availability of TPU-like AI accelerators make our approach highly compelling. Such a gap is largely attributed to control overheads, mismatching HE computation to the compute pipeline in hardware, extreme data movement, and poor data reuse [56], [59], [71].

This paper identifies a compelling alternative: leveraging existing ASIC AI accelerators, such as Google's TPUs, for HE operators to achieve better energy efficiency (performance per watt), because TPU offer *functionally equivalent hardware components as GPUs but employ coarse-grained control to amortize the control overheads* (Fig. 2). However, this coarse-grained design creates a fundamental mismatch with HE algorithms that enables GPU to achieve SoTA throughput. This is because they require high-precision compute and fine-grained data manipulation for each individual data element, and both are inefficient for TPU. Specifically,

Arithmetic Mismatch Challenge: SoTA GPU HE libraries (1) either have an inefficient precision lowering algorithm to

¶We appreciate Arvind's technical feedback and strong encouragement. Unfortunately Arvind passed away before this work is published, but we feel extremely honored to have this paper become a part of his strong legacy.
§This work was done before this author joined NVIDIA.

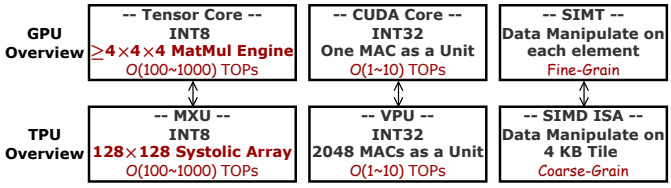


Figure 2: TPU's compute/memory granularity is > GPU.

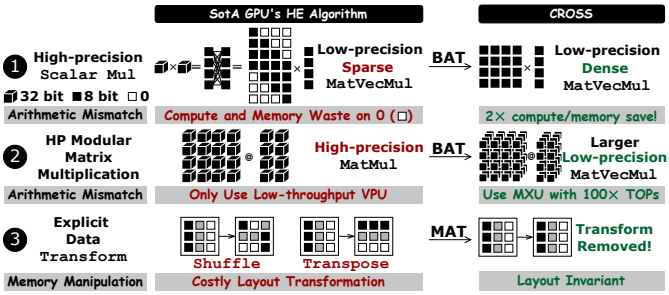


Figure 3: CROSS aims at (1) eliminating compute redundancy, (2) leveraging powerful MXU for throughput improvement, and (3) removing explicit memory costs for better efficiency.

create a sparse matrix with redundant zeros (❶ in Fig. 3 with details in Fig. 7), wasting compute and memory and reducing effective compute utilization of matrix multiplication unit (MXU), (2) or rely on 32-bit integer operations [17] that are available in TPU's low-throughput Vector Processing Unit (VPU), leaving its powerful, high-throughput low-precision 8-bit MXU fully idle (❷ in Fig. 3).

Memory Manipulation Granularity Challenge: HE kernels like the Number Theoretic Transform (NTT) require frequent, fine-grained data shuffling and transposing. These are prohibitively slow on TPUs, because individual data elements need to be fit into large, coarse-grained, and SIMD-controlled (8, 128) 32-bit registers (4 KB VReg) for achieving desired data manipulation. This reduces effective tile utilization (❸).

To tackle above divergence, we propose ***CROSS***[1], a compiler framework that (1) refactors SoTA HE operators into arithmetic that leverage high-throughput MXU with better utilization, and (2) embeds transpose and shuffling into computation in compile time, eliminating runtime data reordering and ensuring a layout-invariant execution. CROSS enables AI accelerators to achieve SoTA energy efficiency (performance per watt) with two architecturally universal optimizations:

Basis Aligned Transformation (BAT) reformulates the sparse left matrix (❶) of preknown parameters (e.g., twiddle factors in NTT and precomputed primes in Basis Conversion) into a dense half-size low-precision matrix vector multiplication, reducing redundant work and memory waste zeros and increasing compute utilization. Further, BAT converts high-precision (e.g., 32-bit) modular matrix multiplication into dense low-precision (8-bit) matrix multiplication (❷).

Memory Aligned Transformation (MAT) eliminates expensive runtime data reordering by embedding layout changes, e.g., transpose and shuffling in NTT, directly into the computation. This is achieved by representing reordering as transformation matrix and applies it to pre-known parameters offline (e.g., twiddle factors in NTT), thus creating a "layout-invariant" kernel with no explicit memory overhead (❸).

To the best of our knowledge, CROSS represents the first work to investigate deployment of HE operators on AI accelerators, introducing a new paradigm of AI/HE co-acceleration on the same hardware substrate. CROSS shows that AI accelerators without any modifications can indeed be used to accelerate HE operators and achieve SoTA efficiency.

Our evaluation on a real single-host Google TPUv6e shows that we can 1.2/13× more throughput of NTT over WarpDrive / TensorFHE+ on NVIDIA-A100, which refreshes the SoTA record of NTT throughput in practically available devices. Furthermore, CROSS achieves 451×, 7.81×, 1.83×, 1.31×, 1.86×, and 1.15× higher throughput per watt on HE operators than OpenFHE, WarpDrive, FIDESlib, FAB, HEAP, and Cheddar, when scaling TPUv6e to consume roughly the same power as their platforms. These results establish AI accelerators (1) the SoTA energy efficient solution among commodity hardware like GPUs and FPGAs, and (2) a promising platform to investigate for privacy-preserving computation.

The primary performance gap with HE ASICs (3-33×) stems from (1) the absence of low-cost data shuffling engine, (2) no hardware support of carefully selected moduli, and (3) less overall memory/compute, as discussed in §V-G.

We make the following contributions in this paper:

- A systematic characterization of the architectural mismatches and inefficiencies that arise when running SoTA GPU-based HE algorithms on AI accelerators like Google TPU.
- Basis-Aligned Transformation (BAT), a novel method to map high-precision modular integer arithmetic to low-precision matrix multiplication engines, enabling high-throughput low-precision matrix multiplication engine in AI accelerators to be used efficiently for HE operators.
- Memory-Aligned Transformation (MAT), a technique to create layout-invariant HE operators that eliminate explicit data transpose and reordering overhead by embedding these operations into computation.
- A comprehensive evaluation demonstrating that CROSS on TPUs achieves SoTA NTT throughput and SoTA energy efficiency for critical HE operators compared to highly optimized CPU, GPU, and FPGA implementations.

II. Background and Motivation

In this section, we introduce the background for Homomorphic Encryption (HE) and discuss key potential of AI accelerators for HE acceleration.

A. Homomorphic Encryption Background

HE is a specialized form of public-key encryption that enables computations to be performed directly on encrypted data without revealing the underlying plaintext. In an HE system, the client exclusively holds a private decryption key, while a public encryption key is made available for data encryption. Additionally, an evaluation key is provided to the cloud or computing service to facilitate computations on the

[1]Compiling Real-time Online Secure Service on ASIC AI Accelerators

Table I: Notations

Term	Conditions	Meaning (example value)
N	Power of two	Polynomial degree (2^{16})
Q		Ciphertext modulus (1728 bits)
q_i	coprime	RNS base $Q = \Pi_{(i=0)}^{(L-1)} q_i$ (28 bits)
L	$\log_2(Q)/\log_2(q_i)$	The number of limbs ($\lceil 1728/28 \rceil$ limbs)
L'		number of limbs with auxiliary modulus
$\log_2 q$		RNS bases have $\log_2 q$ bits, noted as $\log_2 q$
bp		*B*it *P*recision of MAC in hardware
ω_n		A primitive n-th root of unity
$\mathbf{a}, \mathbf{b}, \mathbf{z}$		Coefficient vector
$dnum$		Number of digits in the switching key
K		Number of 8-bit chunks in high-precision scalar
$\mathscr{B}$	$\mathscr{B} = q_0, \cdots, q_{L-1}$	A set of RNS bases.

encrypted data. In HE, direct computation on the encrypted data will apply computation on the underlying messages, such that the entire computation is secured in the privacy-preserving manner as shown in Fig. 1.

*1) **Terminology and Data Representation**:* The security of HE schemes is based on the hardness of Ring Learning With Errors (RLWE) problem [48]. This is a problem over a polynomial ring $R_Q := \mathbb{Z}_Q[x]/(x^N+1)$, where N is a power of two. An element in R_Q is a polynomial of the form $a(x) = \Sigma_{j=0}^{N-1} a_j \cdot x^j$, where each coefficient a_j is an integer in $[0, Q-1]$ and the polynomial is reduced by x^N+1.

*2) **Parameter Determination**:* (degree N, ciphertext modulus bitwidth $\log_2 Q$) are design choices, which determine the security level for a given error standard deviation. Practical applications typically require 128-bit security level, which comes in various choices from (2^{10}, 29) [7] to (2^{17}, 2200+) [59]. A larger coefficient modulus ($\log_2 Q$) allows more computation on a ciphertext before bootstrapping is required, but the degree N of the polynomial modulus must grow with $\log_2 Q$ in order to maintain security. Larger Q and higher degree both lead to longer computational latency. Therefore, the minimal (degree, $\log_2 Q$) that satisfies the required computation is often selected to minimize overheads when bootstrapping is not required.

*3) **Residue Number System (RNS)**:* Once (N, $\log_2 Q$) is being made, each raw data will be encoded and encrypted into a ciphertext with a pair of polynomials of degree N. Each coefficient would be high-precision, often thousands of bits [7], [59], which are not natively supported by the 32/64-bit micro-architectures in CPUs or GPUs. Naively, mapping high-precision data to low-precision computation unit requires two steps: (1) breaking high-precision coefficients into low-precision chunks supported by computation units and (2) executing multiplicative operations across all pairs of chunks from two coefficients. This segmentation and *chunkwise multiplication* incur *quadratic* computational overheads.

To reduce such quadratic pair-wise computation costs, the Chinese Remainder Theorem (CRT) allows us to construct a set of coprime RNS basis $\{q_0, \cdots, q_{L-1}\}$, where $Q = \Pi_{i=0}^{L-1} q_i$. Under CRT, each high-precision coefficient $a_n, n \in [0, N-1]$ of a polynomial in R_Q is represented as residues of a sequence of L smaller moduli $\{a_n \bmod q_i\}$ for $i \in [0, L-1]$. These L obtained polynomials with low-precision coefficients are referred to as limbs, noted as ($limb_i$), $i \in [0, L-1]$. The isomorphism $R_Q \cong R_{q_0} \otimes \ldots \otimes R_{q_{L-1}}$ allows addition and multiplication to be performed "limb-wise" over elements of R_Q. Thus, a limb of one polynomial multiplies only with its counterpart limb of another polynomial, reducing quadratic computational overhead down to linear. In the post-CRT ciphertext, different limbs get processed independently, facilitating limb-level parallelism. A summary of the notations is listed in Tab. I.

However, RNS cannot directly reduce high-precision data to an arbitrary low precision due to inherent constraints. Specifically, RNS requires a set of moduli that (1) multiply to the original modulus Q, and (2) are pairwise co-prime. On typical AI accelerators [32], [34], the lowest supported precision is 8-bit integer arithmetic. To reduce 2000-bit data to 8 bits, $2000/8 = 250$ primes are needed. However, it is infeasible to find 250 co-prime integers within the 8-bit range $[0, 256)$. Consequently, precision reduction to 8-bit arithmetic involves two stages:
(1) *Linear Precision Reduction*: RNS lowers high precision to an intermediate precision (e.g. 32-bit) with linear complexity.
(2) *Quadratic Precision Reduction*: A secondary reduction further lowers the intermediate precision to 8-bit arithmetic, incurring quadratic complexity, *the costs optimized by CROSS.*

*4) **Compute and Memory Analysis**:* Above RNS based encoding and encryption introduce about 200× data expansion of the original message, making HE operators memory-bound for devices with small on-chip memory (*memory overhead*) [22], [35]. Further, evaluating encrypted data also introduces extra computational complexity, i.e. raw multiplication becomes multiplication of multiple high-degree polynomial rings in HE (*computation overhead*). Both overheads lead to significantly long wall-clock time latency of HE-based privacy-preserving serving. Fortunately, such significantly high compute and memory exhibit inherent independence across ciphertexts, polynomial degree (N) and limbs (L) etc. Such independence brings high parallelism (eg. up-to $N \cdot L \approx 2^{22}$ for a single ciphertext multiplication), offering the possibility of hardware acceleration with sea of compute to reduce latency.

B. Potential of AI Accelerators for HE Workloads

AI accelerators feature with abundant memory and compute (Fig. 4), which show great potential for HE acceleration.

• **Large Compute Array (Parallelism):** Each MXU in AI accelerators is 32× larger (e.g., 128×128 in TPUv4) than those in GPUs (typically 4,4,4-Matrix Multiplication [19]). The larger size of MXUs increases on-chip data reuse within the two-dimensional computation arrays, enhancing throughput per watt. Moreover, a sea of 2048 SIMD ALUs, sharing the same VRegs with MXUs and further increasing data reuse.

• **Large On-chip Memory:** AI accelerators feature large on-chip memory, e.g. a single Google TPU v4 chip has 160 MB of on-chip memory, including 128 MB CMEM and 32 MB VMEM array in Fig. 4, which is 20×/4× larger than the AMD MI100 / NVIDIA A100. This substantial on-chip capacity can accommodate entire ciphertexts to avoid its repeated accesses from off-chip memory, alleviating memory bottleneck inherent in HE workloads for better performance and efficiency.

• **On-chip Data Management Units:** TPU has specialized Cross Lane Unit (XLU) in Fig. 4, which could (1) transpose data sitting in on-chip VMEM, (2) shuffle data among

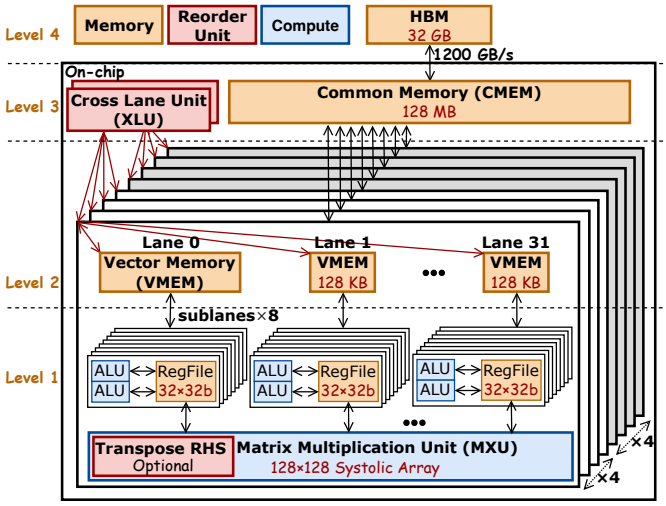


Figure 4: Overview of TPUv4 architecture based on public information [28], [32]–[34]. Four black and gray boxes represent two tensor cores, separately. Two tensor cores share the same 128 MB common memory (CMEM, removed in newer TPUs) to hold frequently used data. Each tensor core has 4 matrix multiplication units (MXU) and 2048 ALUs in Vector Processing Unit (VPU) organized as 128 SIMD lanes. Each lane consists of 8 SIMD sublanes with 128 KB vector memory (VMEM), each sublane has 2 dual-issue ALUs and 128 B local register file. Such two level of SIMDs force a group of (8, 128) 32-bit registers, termed as VReg, to be operated in the lock step. Each MXU features a 128×128 systolic array (256×256 for TPUs after v6) for performing matrix multiplication. Each MXU has a local transpose unit to optionally transpose right-hand-side (RHS) input matrix in the pipelined manner to hide transpose latency behind. Data in VMEM of different lanes could get transposed or shuffled or accumulated through the Cross Lane Unit (XLU), which consumes non-hidden layout reordering and reduction latency.

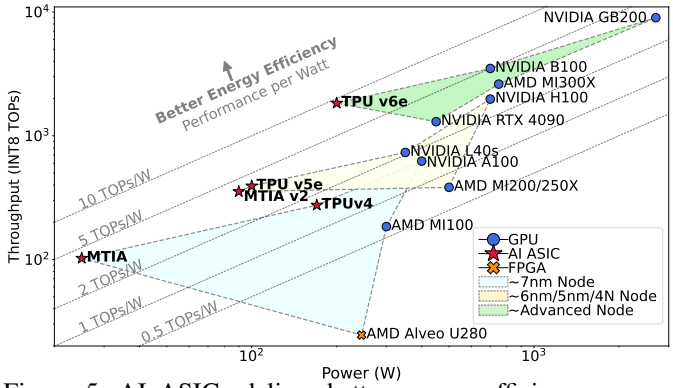


Figure 5: AI ASICs deliver better energy efficiency among practical devices using the same technology nodes.

Table II: Architectural comparison of AI accelerators in Fig. 5.

	TPU	GPU	FPGA
clock frequency	O(1) GHz	O(1) GHz	O(300) MHz
on-chip memory	O(100) MB	O(40) MB	O(40) MB
Programming	SIMD	SIMT	Bits-level Reconfiguration
INT8 TOPs	O(1000)	O(1000)	O(25)

VMEMs, and (3) accumulate partial results from VMEMs in different lanes into final results.

In summary, *large compute array, large on-chip memory and flexible permutation/transpose/reduction engine* enable AI accelerators to achieve better energy efficiency (*performance per watt*, measured by TOPs/watt) over CPU, GPU and FPGAs (see Fig. 5). However, effectively leveraging these abundant resources to accelerate HE remains a challenge due to the misalignment between compute patterns of SoTA HE algorithms and architectural capabilities of AI accelerators.

The core strategy of CROSS is to systematically remap HE operations to align with the architectural strengths of AI accelerators. CROSS converts high-precision modular arithmetic, the essential computational kernel of HE operators, into dense, low-precision matrix multiplications (`MatMul`). This transformation is designed to simultaneously embed costly data reordering (transpose and shuffling) directly into the computation, making layout invariant throughput the computation. By doing so, CROSS enables HE workloads to harness the high-throughput matrix engines of AI accelerators like MXU in TPU while sidestepping the performance bottlenecks associated with fine-grained data movement, achieving SoTA energy efficiency on existing AI hardware without any modifications.

With this aim, we first characterize the performance inefficiency when porting SoTA GPU-optimized HE algorithms to AI accelerator using Google's TPU as an example in §III. To resolve the inefficiency in compute and memory, separately, we propose key two ingredients of CROSS, *Basis Aligned Transformation (BAT)* and *Memory Aligned Transformation (MAT)* in §IV-A and §IV-B. Detailed CROSS evaluation and comparison to SoTAs are analyzed in §V.

III. Inefficiency of SoTA HE Alg. for AI Accel.

While GPUs currently deliver SoTA performance for accelerating Homomorphic Encryption (HE) workloads [17], their architectural characteristics differ fundamentally from ASIC-based AI accelerators, such as Google TPUs (Tab. II). This architectural divergence leads to significant inefficiencies when directly porting GPU-optimized HE algorithms onto TPUs. To systematically analyze these inefficiencies, we first abstract SoTA HE acceleration techniques into distinct compilation layers and explicitly define the scope targeted by CROSS. Within this defined scope, we identify and characterize the key sources of performance degradation in GPU-tailored HE kernels on TPUs, motivating a novel compilation approach tailored specifically for efficient HE on AI accelerators.

A. Optimizations Overview of HE Acceleration Stack

HE acceleration techniques, particularly for CKKS encryption, could be formally categorized into five distinct layers, including Packing, Mapping, Scheduling, Decomposing and Binding, as illustrated in Fig. 6.

• **Packing** defines how data is organized within ciphertext slots. Specifically, a CKKS ciphertext operates like a vector of SIMD units, encoding multiple data per ciphertext and enforcing lock-step operator for all encoded data. Thus, operators in original application initially designed for element-wise computations must be transformed into SIMD-compatible, HE-specific Privacy-Preserving Operators (PP-Ops).

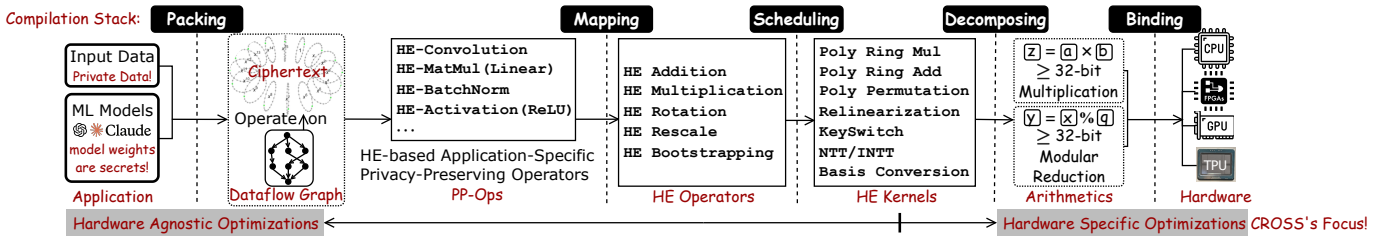


Figure 6: Abstract compilation layers for accelerating HE-based privacy-preserving applications.

• **Mapping** translates PP-Ops into sequences of fundamental HE operators (see Fig. 6). Optimal mapping seeks maximum arithmetic intensity, data reuse, and parallelism while minimizing computational and memory overhead to reduce latency. Application-specific optimizations typically reside here [16], [23], [36], [43]–[45]. Each HE operator consists of multiple HE kernels.

• **Scheduling** determines how HE kernels are scheduled for each HE operator (e.g., addition, multiplication, rotation, rescale, bootstrapping). Effective scheduling algorithms aim at reducing kernel invocation counts, saving latency [9].

• **Decomposing** specifies arithmetic and memory operations on individual ciphertexts for HE kernels. For example, NTT algorithms like radix-2 Cooley-Tukey [17] and 4-step NTT [26] are different algorithms in the decomposing layer.

• **Binding**: Arithmetic and memory operations are translated into hardware-specific programming interfaces (e.g. JAX for TPU), or low-level hardware ISAs (e.g. SIMD ISA for TPU).

Optimizations in packing, mapping and scheduling layers are hardware-agnostic, universally beneficial across hardware platforms. While optimizations in decomposing and binding algorithms are hardware-specific to improve the performance of HE kernels on a specific platform. CROSS primarily addresses inefficiencies in these hardware-specific optimizations for GPU when ported to TPU-like accelerators. We specifically highlight GPU's HE optimizations due to their SoTA performance and hardware similarities with AI accelerators.

B. Architectural Differences: GPU vs. TPU

*1) **Arithmetic Mismatch - Ratio of MatMul to VecMul Throughput**:* While both GPUs and TPUs feature 32-bit integer vector multiplication units (VPUs/CUDA cores) and 8-bit matrix multiplication units (MXUs/Tensor cores), their throughput ratios differ significantly. For instance, one NVIDIA A100's 8-bit Tensor core offers approximately 4× the throughput of a single 32-bit integer CUDA core (one 32-bit integer MAC is approximated as 16 8-bit MACs), limiting performance gains from vector-to-matrix transformations. Conversely, TPU's MXUs achieve substantially higher throughput relative to VPUs (e.g., 58× on TPUv4). This creates a massive incentive to reformulate HE algorithms to leverage the MXU, even if the conversion introduces extra computational overhead.

*2) **Memory Manipulation Granularity - Fine-grained vs Coarse-grained**:* GPUs feature fine-grained, per-core registers optimized for efficient, element-wise data manipulation. Conversely, TPUs prioritize energy efficiency through a coarse-grained register architecture, consisting of a unified large register file (4 KB) shared across multiple VPUs and MXUs. This design significantly reduces instruction overhead but imposes considerable costs for fine-grained, element-wise data manipulations due to low utilization when mapping small data elements to large registers. Consequently, TPUs inherently favor algorithms that leverage large-tile SIMD computation, while avoiding costly element-wise manipulations.

C. Inefficiency of SoTA Binding Algorithm on TPU

Current GPU-optimized HE kernels predominantly use 32-bit arithmetic operations [17], including scalar multiplication and modular reduction, which are a natural fit for a GPU's CUDA cores. On a TPU, however, this has dire consequences:

1) Idle Matrix Engine for Computing HE Kernels: SoTA GPU implementations perform HE kernels such as Basis Conversion (BConv, details in §F2) and radix-2 Cooley-Tukey Number Theory Transformation (NTT, details in §F1) as sequences of *32-bit scalar multiplications*. On TPUs, these 32-bit multiplications are executed on the low-throughput VPU, leaving the high-performance low-precision MXU idle.

2) Redundant Zeros in 32-bit Mul.: In GPU's implementation, a high-precision multiplication is decomposed into a sparse matrix-vector multiplication (`MatVecMul`) of lower-precision chunks (e.g., 8-bit). As shown in Fig. 7 [26], the post-conversion `MatVecMul` contains a sparse matrix with nearly half zeros, wasting compute and memory resources.

More efficient method is needed to remove these zeros and convert high-precision arithmetic into dense low-precision matrix operations with no redundant compute and memory. This motivates Basis Aligned Transformation (BAT).

D. Inefficiency of SoTA Decomposing Algorithm on TPU

HE kernels are rife with data reordering operations that are fundamentally at odds with the TPU's coarse-grained memory system. The reordering required by two latency dominating HE Kernels is analyzed below. Full profiling results see §F.

1) Number Theoretic Transform (NTT): It has the highest algorithmic complexity and latency among HE operators.

• *Radix-2 Cooley-Tukey NTT algorithm*, optimized for GPUs, relies on fine-grained, bit-complement shuffling at each stage. On a TPU, this requires moving small, non-contiguous data blocks across different memory lanes via the Cross Lane Unit (XLU), resulting in extremely low data tile utilization and prohibitive memory latency. We provide details in §F1.

• *4-step NTT algorithm* reformulates the NTT into a chain of matrix multiplications, making it a better candidate for TPUs.

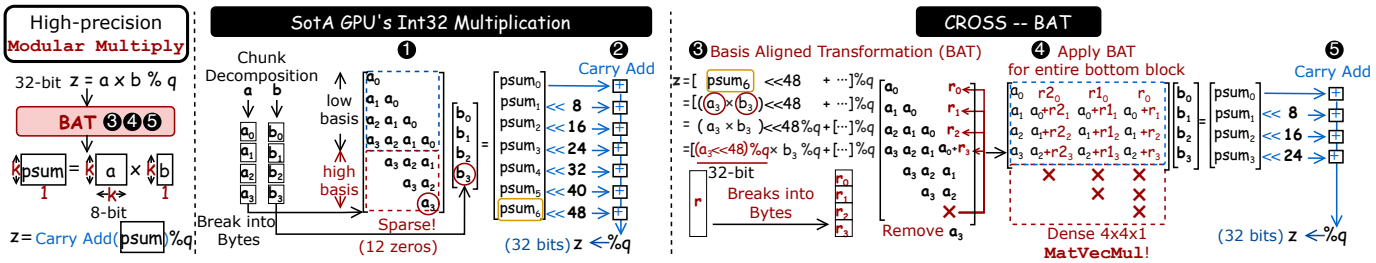


Figure 7: BAT converts high-precision modular scalar multiplication into dense, low-precision matrix multiplication. Compared to the SoTA approach used in GPUs, BAT achieves 2× theoretical computational and memory savings by eliminating redundant zeros of converted matrix. Subscripts indicate chunk indices with each chunk 8-bit long. Modular reduction details in Alg.1.

However, it introduces both matrix transpose and bit-reverse shuffling. This large-scale data reordering is costly on the TPU architecture, worsening memory bottleneck.

2) Automorphism: It requires explicit slot-wise data permutation within a ciphertext, again leading to low VReg utilization and non-trivial memory latency.

These explicit and costly data reordering steps demand a scheduling strategy that removes runtime reordering. It motivates our Memory-Aligned Transformation (MAT).

Post-compiled arithmetic differences among HE workloads, SoTA GPU's HE library and CROSS are listed in Tab. III.

Table III: Arithmetic Comparison (GPU's HE lib. vs. CROSS)

	Primitive	Precision	Operation
HE	`VecModAdd`	$\log_2 q$	$(\mathbf{a}+\mathbf{b}) \bmod q$
	`VecModMul`	$\log_2 q$ ①	$(\mathbf{a}\times\mathbf{b}) \bmod q$
	`ModMatMul`	$\log_2 q$ ②	$M_{H\times V}\cdot M_{V\times W} \bmod q$
Kernels in GPU's HE Library	`VecAdd`	(u)int32	$\mathbf{a}+\mathbf{b}$
	`VecMul`	(u)int32	$\mathbf{a}\times\mathbf{b}$
	Sparse `MatMul`	(u)int8	$M_{(H+K-1)\times V}\cdot M_{V\times W}$
	`Transpose`	32 bit	$M_{H\times V}\rightarrow M_{V\times H}$
	`Permutation`	32 bit	Across Lane (Fig. 4)
CROSS	`VecAdd`	(u)int32	$\mathbf{a}+\mathbf{b}$
	`VecMul`	(u)int32	$\mathbf{a}\times\mathbf{b}$
	Dense `MatMul`	(u)int8	$M_{H\times V}\cdot M_{V\times W}$

①: $28 \le \log q \le 59$ under 128-bit security level; Intermediate results need $2\log q$ bits, far exceeding precision range of GPUs or AI accelerators.
②: intermediate results need $2\log q + \log V$ bits to avoid precision overflow.

IV. CROSS Methods

This section introduces two key contributions, BAT and MAT, in resolving arithmetic mismatch and memory manipulation granularity challenge categorized in §III-B.

A. Basis Aligned Transformation (BAT)

BAT, our innovation in binding algorithm, is designed to efficiently transform high-precision modular integer arithmetic into dense low-precision matrix multiplication, enabling the TPU's MXU, with $\sim O(100)$ times higher throughput than its VPU, to effectively accelerate primitives in Tab. III.

The key idea of BAT lies in offline pre-computing known parameters—such as twiddle factors in NTT, primes in BConv, and evaluation keys in relinearization—to minimize runtime overhead in transformed low-precision matrix multiplication. We first illustrate BAT's methodology using a concrete example of transforming 32-bit integer multiplications into efficient 8-bit `MatVecMul`. We then highlight BAT's advantages over the SoTA 32-bit multiplication approach used in GPUs and explain its applicability to enhancing Montgomery reduction and high-precision Modular matrix multiplication (`MatModMul`).

1) BAT Methodology and Illustration: In the SoTA high-precision scalar multiplication used by GPUs [26], [63], two inputs of 32-bit standard modular multiplication, a and b, are first broken into four 8-bit chunks (a_0,a_1,a_2,a_3) and (b_0,b_1,b_2,b_3). Then it performs all-to-all chunk multiplications to produce products (termed as partial sum, noted as psum) that contribute to different "output bases" (i.e., powers of 2^8). For instance, $a_3\times b_3$ contributes to the final sum with a basis of 2^{48}. The SoTA approach maps the computation from input chunks to partial sums as a sparse `MatVecMul`, as shown in Fig. 7 (❶), where left matrix is a toeplitz matrix of all chunks of a. It contains $12/(4\times 7)\approx 43\%$ zeros and leads to redundant compute and memory overhead. Further, it explicitly computes all seven partial sums and accumulates them via a long chain of carry-and-add operation (❷).

BAT recognizes a critical opportunity: since the final result must be taken modulo q, the contributions of high-basis terms can be calculated and "folded" into the low-basis terms in compile time, i.e. BAT applies modulo q to the shifted sparse left matrix, which converts it into a smaller, dense matrix that directly computes the final low-basis coefficients.

Specifically, elements in each left matrix row (❶) contribute to distinct bit ranges in the final 64-bit partial sum (*psum*). If we use bp as bit precision of matrix multiplication, which is 8 for TPU, then each row $k\in[0,6]$ has an associated bit range from $8k$ to $8(k+2)\cdot bp+2$, with an output basis of $2^{k\cdot bp}$. Post modular reduction, the final result z (❷) retains only the four lowest bases (lower 32 bits, for $k\in[0,3]$), rendering higher-basis contributions intermediate. BAT directly applies modular reduction to elements in sparse left matrix contributing to these higher-basis partial sums, realigning them to the lowest bases. For instance, in the case of $a_3\times b_3 = psum^6$, BAT transforms $psum^6$ by computing $r = (a_3 << 48) \mod q$ and subsequently decomposing it into four 8-bit chunks (r_0,r_1,r_2,r_3). These chunks are then added back to the top blue dash box (❸).

By systematically applying BAT to all elements in the block of high basis (red dash box), the sparse left matrix shrinks into a smaller dense matrix (❶→❹), achieving a $\sim 2\times$ theoretical computational and memory saving in the `MatMul`. Further, the length of carry-add chain is reduced from seven to four (❷→❺), saving latency when being mapped to VPU.

Overall, BAT adds an offline pre-computation overhead to eliminate runtime redundancy and reduce the size of temporal reduction, enhancing computation and memory efficiency. The above procedures are illustrated in Fig. 7 with detailed explanation detailed in Alg. 5 in the appendix.

2) Math of BAT: For arbitrary precision input value a, b, each with K bytes. Assuming the value of a is preknown (e.g. twiddle parameters, evaluation key, parameters in basis switch), modular multiplication of a and b is reformed as:

$$a \times b \bmod q = \left(\sum_{i=0}^{i=K-1} \underbrace{a}_{\text{K bytes}} \times \Big(\underbrace{b_i}_{\text{one byte}} \times 2^i \Big) \right) \bmod q \tag{1}$$

$$= \left(\sum_{i=0}^{i=K-1} \underbrace{(a \times 2^i \bmod q)}_{\text{calculated offline as } a_i \text{ } (K \text{ bytes})} \times \underbrace{b_i}_{\text{one byte}} \right) \bmod q \tag{2}$$

$$= \left(\sum_{i=0}^{i=K-1} \underbrace{a_i}_{K \text{ bytes}} \times \underbrace{b_i}_{\text{one byte}} \right) \bmod q \tag{3}$$

$$= \left(\sum_{i=0}^{i=K-1} \underbrace{\Big(\sum_{j=0}^{j=K-1} a_{j,i} \times 2^{8j} \Big)}_{\text{decomposes } a_i \text{ as bytes}} \times \underbrace{b_i}_{\text{one byte}} \right) \bmod q \tag{4}$$

$$= \left(\sum_{j=0}^{j=K-1} \underbrace{\Big(\sum_{i=0}^{i=K-1} a_{j,i} \times b_i \Big)}_{\text{8-bit matrix multiplication}} \times 2^{8j} \right) \bmod q \tag{5}$$

$$= \sum \left(\underbrace{\begin{bmatrix} a_{0,0} & \cdots & a_{0,K-1} \\ a_{1,0} & \cdots & a_{1,K-1} \\ \vdots & \ddots & \vdots \\ a_{K-1,0} & \cdots & a_{K-1,K-1} \end{bmatrix}}_{K \times K \text{ 8-bit matrix}} \times \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{K-1} \end{bmatrix}}_{K \times 1 \text{ 8-bit}} \times \begin{bmatrix} 2^0 \\ 2^8 \\ \vdots \\ \vdots \end{bmatrix} \right) \bmod q \tag{6}$$

$$= \left(\sum_{j=0}^{j=K-1} \Big(\underbrace{psum_j}_{16+\log_2(K) \text{ bits}} \times 2^{8j} \Big) \right) \bmod q \tag{7}$$

BAT offline applies modular reduction to preknown K-byte parameter a in Eq. (1), converting it into Eq. (3). BAT then schedules post converted computation as a low-precision matrix multiplication with a carry propagation. This conversion takes $O(N)$ time for converting a matrix with $O(N)$ elements.

3) Accelerating Core HE Kernels with BAT:

a) BAT Lazy Modular Reduction for Scalar: BAT can be applied to modular reduction by transforming it into a matrix multiplication (MatMul) followed by carry propagation (§J). However, the resulting MatMul has reduction dimension K, which favors small matrix engine like the tensor cores in GPUs but underutilizes the MXU. Therefore, we adopt optimized Montgomery reduction (Alg. 1) and Barrett reduction (Alg. 4) and map both to the VPU.

Algorithm 1 Optimized Montgomery Reduction (64→32 bits)

Require: $z \in [0, 2^{64})$, $q < 2^{32}$, $q_{lo} = q \pmod{2^{16}}$, $q_{hi} = \lfloor q/2^{16} \rfloor$
Ensure: $B \equiv z \cdot 2^{-32} \pmod q$, $B \in [0, 2q)$
1: $z_{lo} \leftarrow z \pmod{2^{32}}$, $z_{hi} \leftarrow \lfloor z/2^{32} \rfloor$ ▷ Split 64-bit input
2: $t \leftarrow (z_{lo} \cdot q^{-1}) \pmod{2^{32}}$ ▷ Low 32-bit product
3: $t_{lo} \leftarrow t \pmod{2^{16}}$, $t_{hi} \leftarrow \lfloor t/2^{16} \rfloor$ ▷ Split t for 16-bit mults
Compute upper 32 bits of $(t \cdot q)$ using 16-bit primitives:
4: $p_{hi} \leftarrow t_{hi} \cdot q_{hi}$; $p_{lo} \leftarrow t_{lo} \cdot q_{lo}$; $p_{m,hi} \leftarrow t_{hi} \cdot q_{lo}$; $p_{m,lo} \leftarrow t_{lo} \cdot q_{hi}$
5: $mid_{lo} \leftarrow p_{m,hi} + p_{m,lo} + \lfloor p_{lo}/2^{16} \rfloor$
6: $mid_{hi} \leftarrow \lfloor p_{m,hi}/2^{16} \rfloor + \lfloor p_{m,lo}/2^{16} \rfloor + \lfloor mid_{lo}/2^{16} \rfloor$
7: $t_{final} \leftarrow p_{hi} + mid_{hi}$ ▷ This equals $\lfloor (t \cdot q)/2^{32} \rfloor$
8: $B \leftarrow z_{hi} + q - t_{final}$ ▷ Result in $[0, 2q)$
9: **return** $B \pmod{2^{32}}$

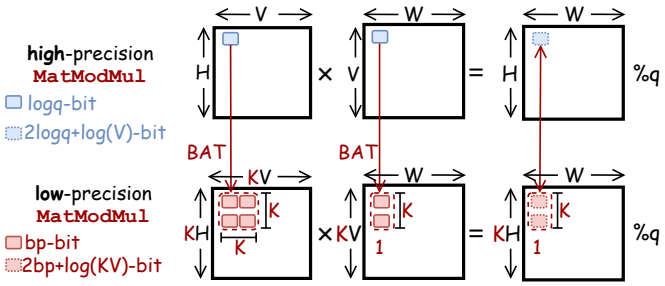


Figure 8: BAT could be applied to convert each individual elements of high-precision `MatModMul` to transform it into low-precision `MatMul`. This enlarges dimensions by a factor of $K = \lceil \frac{\log_2 q}{bp} \rceil$, enhancing MXU utilization. Specifically, CROSS employs BAT to represent each high-precision ($\log_2 q$-bit) element from the input matrices as either a $K \times K$ array or a K-length vector composed of lower-precision (bp-bit) elements. The resulting element has $2bp + \log_2(KV)$ bits, accounting for precision expansion during reduction (Alg. 2).

b) High-Precision `MatModMul` in BConv: BConv requires `VecModMul` and `MatModMul` in two steps (details in §F2). 32-bit `ModMatMul` in step 2 dominates latency in BConv. Abstractly, this operation can be represented as an 32-bit (H, V, W)-`ModMatMul`, which could only utilize the low-throughput VPU, failing to exploit inherent matrix multiplication data reuse, thus limiting performance. BAT addresses this limitation by reformulating the operation into an 8-bit (KH, KV, W)-`ModMatMul` (Alg. 2), suitable for MXU acceleration. This transformation (1) improves the throughput and (2) leverages the inherent data reuse capability of MXU to reduce memory traffic, mitigating memory boundness.

B. Memory Aligned Transformation (MAT)

MAT is a scheduling optimization that removes explicit runtime memory reordering overhead by *integrating data transformation directly into computation*. Its core principle is to permute parameter offline in compile time, thereby ensuring runtime computations inherently produce outputs in

Algorithm 2 Applying BAT to High-Precision ModMatMul

Require: Modulus q; Preknown matrix $A \in \mathbb{Z}_q^{H\times V}$, input data matrix $B \in \mathbb{Z}_q^{V\times W}$; bp: MXU bit precision; $K \leftarrow \lceil \frac{\log_2 q}{bp} \rceil$.
Ensure: $Z \in \mathbb{Z}_{q^2}^{H\times W}$ (high-precision MatMul result).

ChunkDecompose$(a) \rightarrow [a_k]_{0\le k<K}$
for $k = 0$ **to** $K-1$:
  $a_k \leftarrow (a \gg (k\cdot bp))$ & $(2^{bp}-1)$ ▷ Mask k-th bp bits.
Return $[a_k]_{0\le k<K}$

ChunkMerge$([a_k]_{0\le k<K}) \rightarrow a$
$a \leftarrow 0$
for $k = 0$ **to** $K-1$:
  $a \leftarrow a + (a_k \ll (k\cdot bp))$
Return a

DirectScalarBAT$(a) \rightarrow M_{K\times K}$
Initialize $M \leftarrow \mathbf{0}^{K\times K}$
for $j = 0$ **to** $K-1$: ▷ Iterate columns (input basis $2^{j\cdot bp}$)
  $val \leftarrow (a \ll (j\cdot bp)) \pmod q$ ▷ Shift and reduce
  $[chunk_0, \ldots, chunk_{K-1}] \leftarrow$ **ChunkDecompose**(val)
  for $i = 0$ **to** $K-1$: ▷ Fill rows (output basis)
    $M[i,j] \leftarrow chunk_i$ ▷ Assign i-th chunk to row i
Return M

OfflineCompileLeft$(A_{H\times V}) \rightarrow A_{dense}$ ▷ Offline
Initialize $A_{dense} \leftarrow \mathbf{0}^{KH\times KV}$
for $h = 0$ **to** $H-1$:
  for $v = 0$ **to** $V-1$:
    $M_{sub} \leftarrow$ **DirectScalarBAT**$(A[h,v])$
    $A_{dense}[hK:(h+1)K,\ vK:(v+1)K] \leftarrow M_{sub}$
    ▷ Embed $K\times K$ block
Return A_{dense}

RuntimeCompileRight$(B_{V\times W}) \rightarrow B_{dense}$
Transforms Right Matrix to $KV\times W$ layout Fig. 8.
Initialize $B_{dense} \leftarrow \mathbf{0}^{KV\times W}$
for $v = 0$ **to** $V-1$:
  for $w = 0$ **to** $W-1$:
    $[b_0, \ldots, b_{K-1}] \leftarrow$ **ChunkDecompose**$(B[v,w])$
    for $k = 0$ **to** $K-1$:
      $B_{dense}[vK+k, w] \leftarrow b_k$ ▷ Stack chunks vertically
Return B_{dense}

Main-FullMatMul$(A,B) \rightarrow Z$
$A_{dense} \leftarrow$ **OfflineCompileLeft**(A) ▷ $KH\times KV$
$B_{dense} \leftarrow$ **RuntimeCompileRight**(B) ▷ $KV\times W$
$Z_{chunk} \leftarrow A_{dense} @ B_{dense}$ ▷ Low-precision MatMul (MXU)
Initialize $Z \leftarrow \mathbf{0}^{H\times W}$
for $h = 0$ **to** $H-1$:
  for $w = 0$ **to** $W-1$:
    $Z[h,w] \leftarrow$ **ChunkMerge**$([Z_{chunk}[hK,w], \ldots, Z_{chunk}[(h+1)K-1, w]])$
Return Z

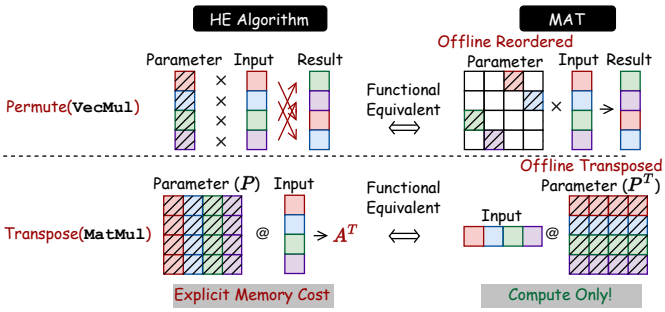


Figure 9: MAT Illustration for `Permute(VecMul)` and `Transpose(MatMul)`. MAT moves explicit memory reordering to compiler time by applying reordering directly on preknown parameters offline for runtime latency saving.

the desired data layouts. This strategy effectively eliminates runtime memory reordering costs.

1) MAT Key Idea and Illustration: MAT leverages the insight that any reordering operation on a one-dimensional vector can be represented as multiplication with a "permutation matrix", which refers to a matrix containing exactly one non-zero element per row and per column. By applying the permutation matrix offline to pre-known parameters (e.g., twiddle factors), the post MAT kernel generates the output in the expected order without incurring runtime reordering. We demonstrate MAT's efficacy using two representative HE workloads: `VecMul`, which requires element permutation, and `MatMul`, which necessitates matrix transposition (Fig. 9). MAT is also effective for multi-dimensional tensors by creating one permutation matrix per dimension offline. We fuse all permutation matrices into one.

2) Applying MAT to 4-step NTT for Layout Invariance:

a) Transpose Elimination: The SoTA GPU's tensor core centric 4-step NTT [24] consists of reforming N-input data into (R,C) matrix and performing four steps:

- Perform R-input NTTs on each of the C columns.
- Transpose resulting $R\times C$ matrix to obtain a $C\times R$ layout.
- Multiply by precomputed twiddle factors.
- Perform C-input NTTs on each of the R rows.

4-step NTT transposes input data from row-major layout into column-major layout, incurring explicit memory overhead. To eliminate it, we apply *MAT* to embed the transpose into the second matrix multiplication as shown in Fig. 9. Leveraging the identity $(A@B)^T = B^T@A^T$, MAT frees the demand of column-major layout back to its input row-major layout, effectively removing transpose. Further, MAT exploits the symmetry structure of twiddle factors (TF), i.e. $(TF^C_{C\times C})^T = TF^C_{C\times C}$). Therefore, MAT merely interchanges the multiplication order of coefficients and twiddle factors of the second matrix multiplication.

b) Bit-reverse Shuffling Elimination: Typical negacyclic NTT produces results in the bit-reverse order. We further apply MAT to embed the bit-reverse reordering into offline coefficients permutation, i.e., permute the rows of Step 1 and Step 2 matrices, and the columns of the Step 3 matrix, using bit-reversal indices. This ensures runtime computation to directly

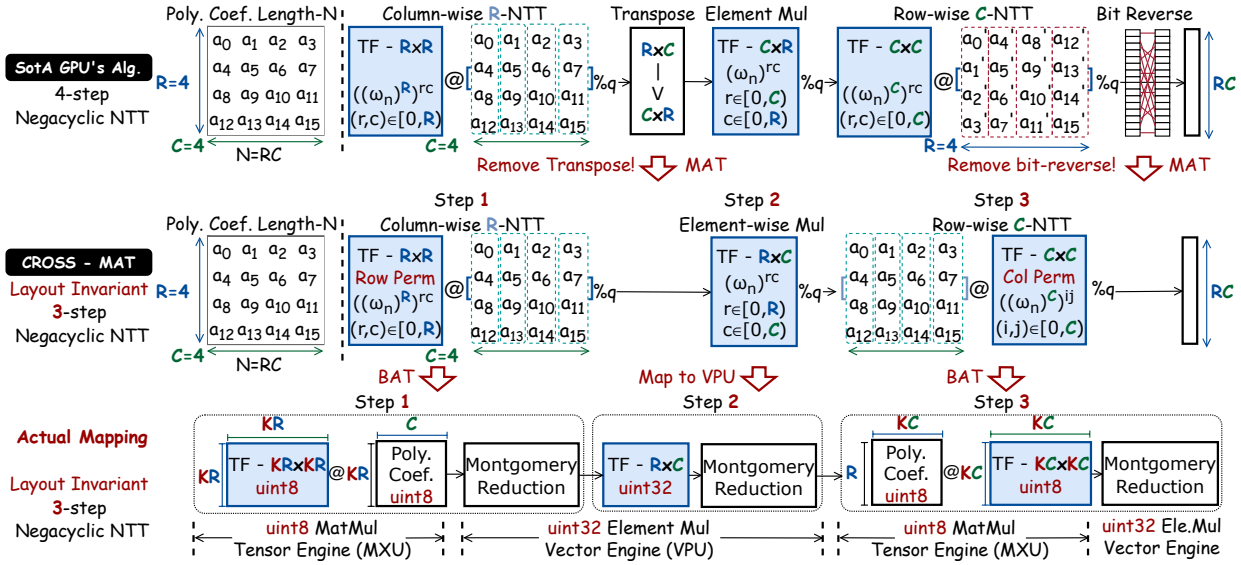


Figure 10: CROSS converts *high-precision NTT* into a combination of low-precision matrix multiplications and element-wise multiplications to fully exploit the MXU and VPU. Row 1 shows the conventional 4-step NTT algorithm [26]; Row 2 illustrates how the Memory Aligned Transformation (MAT) eliminates the explicit transpose and bit-reverse permutation to keep data layout invariant in NTT. Row 3 details the mapping of the MAT-optimized algorithm onto the TPU architecture. Here, TF denotes twiddle factor. Twiddle factors in step 1 and 3 become the same when $R = C$, such that only one TF matrix is needed.

generate the results in the bit-reversed order, i.e., purely using two matrix multiplications to generate a bit-reverse reordered output. Such offline permutation works for bit-reverse order with power of 2 elements, and being applicable to all NTTs and INTTs used in HE. This step ensures "layout-invariant" throughout the entire NTT/INTT process, thus we call it *layout invariant 3-step negacyclic NTT*, as shown in Fig. 10. It's detailed formula is:

$$\Big(\underbrace{(P_{\pi(R)} @ TF^{R}_{R\times R})}_{\text{offline row permutation}} @ a_{R\times C}\Big) \cdot \underbrace{(P_{\pi(R)} @ TF_{R\times C})}_{\text{offline row perm.}} \Big) @ \underbrace{(TF^{C}_{C\times C} @ P_{\pi(C)})}_{\text{offline column perm.}}$$

Here, @ denotes matrix multiplication and $\cdot$ represents element-wise multiplication. $TF^{k}_{R\times C}$ means a twiddle factor matrix of $[((\omega_n)^k)^{rc}], r \in [0,R), c \in [0,C)$. ω_n: primitive n-th root of unity. $P_{\pi(R)} = \left[\delta_{c,\rho(r)}\right]_{r,c\in[0,R)}$ and $P_{\pi(C)} = \left[\delta_{r,\rho'(c)}\right]_{r,c\in[0,C)}$ denote bit-reversal permutation matrix. $\rho(i)$ and $\rho'(i)$ denote the bit-reversal of index i for size R and C, separately. $\delta_{i,j}$ is the Kronecker delta (1 if $i = j$, 0 otherwise).

c) Mapping to TPU: In layout-invariant 3-step negacyclic NTT, high-precision matrix multiplications in steps 1 and 3 are transformed into low-precision matrix multiplications by BAT, which will be executed by MXU to achieve high throughput. Other operators are mapped to VPU (Fig. 10).

Integrating MAT and BAT, CROSS efficiently exploits both MXU and VPU in sequential order listed from left to right in the third row of Fig. 10. The resulting algorithm achieves an effective computational complexity of $O(N\sqrt{N})$ while completely avoiding costly data reordering. Although this complexity is higher than the radix-2 Cooley-Tukey NTT ($O(N \log N)$), the dramatic throughput advantage of MXU results in superior overall NTT throughput and energy efficiency, making TPUs the SoTA throughput engine for NTT.

V. Evaluation

We investigate the performance benefits of BAT and MAT exclusively for latency dominating HE operators. We also evaluate the throughput per watt (energy efficiency) of CROSS framework on Google's TPU architectures against SoTA implementations on various platforms including CPUs, GPUs, FPGAs, and HE ASICs. We analyze the reasons behind performance gap to the dedicated HE ASICs, and summarize future directions to improve AI ASIC's HE performance.

A. Methodology

We select Google's TPU as a representative ASIC AI accelerator and enable deployment on real TPUs by converting HE operators into `MatMul` and `VecMul` and leveraging JAX [12] to program TPUs. We report the latency obtained from trace viewer in XLA profiler [18]. For bootstrapping, MNIST inference and Logistic Regression, the estimated latency is obtained by multiplying the overall number of HE kernels invocations with each profiled realistic latency, which represents the worst case latency as it assumes no pipeline or fusion.

Security Parameter Selection: Security parameter setup has significant effect on the performance and accuracy. Given TPU's micro-architecture is mainly designed for optimizing performance of low-precision integer (up-to 32 bits) and it implements 32-bit registers, we choose the security parameter with $\log_2 q < 32$ for better performance. For security parameters requiring moduli precision exceeding 32 bits, we employ double rescaling [14] to discard two sub-moduli ($\log_2 q < 32$) per level, doubling the number of constituent moduli.

CROSS Configuration: Unless otherwise stated, all experiments use a 128-bit security configuration with $\log_2 q = 28$, $L = 51$, $D_{\text{num}} = 3$, and $N = 2^{16}$ (Set D in Tab. IV). Each 28-bit coefficient is stored in a 32-bit integer, and decomposed into four 8-bit chunks to match MXU operand precision when aiming at using MXU. CROSS employs a layout-invariant 3-step NTT, with $(R,C) \in \{(128,512),(256,256),(512,128)\}$ selected to efficiently map onto MXU. For standalone NTT evaluation, we fix $(R,C) = (128, \lfloor N/128 \rfloor)$, explicitly setting one dimension to 128 (the number of lanes) to ensure full VReg utilization at small problem sizes, such as $\sqrt{N} < 128$. For HE operator evaluation, we sweep three (R,C) configurations and report results using the best-performing one.

Workload: We adopt ML workloads without bootstrapping including MNIST and Logistic Regression[2] [30], and four backbone HE operators as workloads, including HE-Add, HE-Mult, Rescale, and Rotate [15]. CROSS's HE operator implementations are verified against OpenFHE's leveled ckksrns [8], achieving the same accuracy and generality as the OpenFHE's implementation. We adopt the packed bootstrapping algorithm defined in [3].

Baselines: *For TPU*, the baseline refers to using the SoTA GPU's decomposing and binding algorithm, including (1) breaking high-precision modular scalar multiplication as low-precision modular `MatMul` illustrated in Fig. 7 and (2) 4-step NTT. *For other platforms* like CPU [50], GPU [17], [26], FPGA [4], and HE ASIC [27], [56], the security configuration leading to the best performance for each platform is chosen. We further choose the latest works which report NTT throughput and latency of HE kernels in each evaluation, respectively.

AI-ASIC Devices Setup and Metrics. We deploy CROSS on a single-host TPU virtual machine (TPU-VM) of different generations, including TPUv4, TPUv5e, TPUv5p, and TPUv6e. We use TPUv6e by default. The detailed number of tensor core and performance specifications per Tensor Core (TC) are listed in Tab. IV. A TPU-VM refers to a group of TPU chips that share the same CPU host, e.g. four chips being arranged as 2×2 torus to be controlled by a single CPU.

Metric: For fair comparison with prior works [5], [17], [24], [27], [56], we report latency and throughput per watt following methodology below. For *latency* of each HE kernel, we run the same kernel on each tensor core and report amortized single-batch latency. For *throughput per watt* (*energy efficiency*), we measure the number of kernels completed per second under a TC configuration that matches the thermal design power (TDP) of the comparison device. Specifically, we constrain the system to 4 TCs when comparing against the Alveo U280 (FPGA), A100 (GPU), and BASALISC/CraterLake (ASIC) architectures, and scale to 2 TCs for the AMD 9950X3D (CPU), and 8 TCs for RTX 4090 (GPU) baselines and HEAP [2], respectively. We measure device energy efficiency using the average latency of dominating HE-Mult and Rotate.

[2]HELR [30] is a binary classification model using logistic regression. We trained the model for 32 iterations, each with a batch containing 1024 14×14-pixel MNIST images, where an iteration is a gradient update step with a single batch, and report average execution time per iteration.

Table IV: TPU setup and NTT evaluation setup.

Hardware	**TPUv4**	**TPUv5e**	**TPUv5p**	**TPUv6e**
# JAX Logical Devices	4	4	4	8
Setup	v4-8	v5litepod-4	v5p-8	v6e-8
# Tensor Cores	8	4	8	8
GFLOPs	139800	202700	236700	918000
HBM BW (GiB/s)	572	763	1287	1526
VMEM Read BW (GiB/s)	2003	17166	20027	21696
VMEM Write BW (GiB/s)	1001	5722	6676	15020
Configuration	$\log_2 Q$	Degree	$\log_2 q$	#Limbs
Set A	109	2^{12}	28	4
Set B	218	2^{13}	28	8
Set C	438	2^{14}	28	15
Set D	1904	2^{16}	28	51

TPUv6e (Set D) is the default configuration used by CROSS in evaluations. Specifications (FLOPs and BW) obtained from XProf for **one tensor core**. TPUv6e offers higher GFLOPs as it has 256×256 systolic array.

B. Evaluating Performance of Individual CROSS Optimization

1) BAT Evaluation:

a) High-precision `MatModMul` – BAT vs. baseline: `MatModMul` in SoTA GPU's HE algorithm mainly arises from NTT and INTT. NTT-based `MatModMul` typically involves square matrices of size 2^8 to 2^{11}. On a single TPUv6e Tensor Core, BAT achieves a speedup of up to 1.62× over the low-precision sparse matrix multiplication baseline. This efficiency gain is attributed to the elimination of redundant computations on zero elements and the removal of data-type conversion overhead for static input parameters (e.g., twiddle factors). The lower speedup observed for smaller matrix dimensions is due to their memory-bound nature. In this regime, performance for both BAT and the baseline is constrained by HBM bandwidth rather than peak computational throughput.

Table V: BAT vs. baseline on $M_{H\times V} @ M_{V\times W} \mod q$

H	V	W	Baseline	BAT	speedup
512	256	256	6.00 μs	4.57 μs	**1.31**×
1024	256	256	9.40 μs	6.88 μs	**1.37**×
2048	256	256	15.43 μs	11.06 μs	**1.39**×
4096	256	256	29.09 μs	20.14 μs	**1.44**×
1024	512	512	20.58 μs	16.32 μs	**1.26**×
2048	512	512	38.49 μs	28.48 μs	**1.35**×
1024	1024	1024	59.13 μs	40.69 μs	**1.45**×
2048	1024	1024	113.91 μs	81.71 μs	**1.39**×
2048	2048	2048	365.28 μs	224.80 μs	**1.62**×

b) `BConv` – BAT vs. TPU baseline: Leveraging BAT, the step 2 of BConv is converted from high-precision `VecModMul` into low-precision skewed `MatModMul`, with one dimension equal to degree N and the other being the number of limbs KL or KL' in Tab. VI, resulting in (N, KL, KL')-`MatModMul`. When the reduction dimension in MatMul is not divisible by 128, CROSS pads zeros, leading to partial MXU utilization. Such conversion of using high-throughput MXU brings $\leq 7.16\times$ speedup on one tensor core (Tab. VI).

Table VI: BConv Evaluation (w/ vs. w/o BAT, unit: μs)

limb_in l	limb_out l'	Degree N	Baseline	BAT	Speedup
12	28	65536	815.28	135.91	**6.00**×
12	36	65536	1054.89	147.28	**7.16**×
16	40	65536	165.18	65.77	**2.51**×
24	56	65536	318.92	94.67	**3.37**×

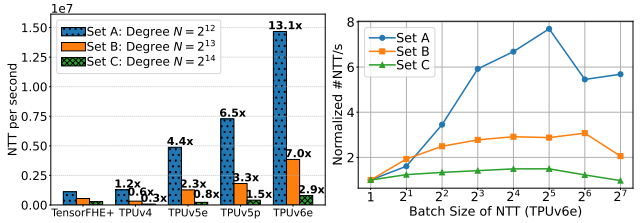
2) MAT + BAT Evaluation – NTT: NTT is the dominant latency kernel in HE serving, so accelerating NTT directly improves overall performance. Under degree $N = (2^{12}, 2^{13}, 2^{14})$ with $\log_2 Q = (109, 218, 438)$ (Tab. IV), CROSS enables TPU

v6e-8 to achieve up-to 99×, 4×, 2×, 13.1×, 1.2× higher throughput over HEAX, FAB (U280 FPGA), HEAP (U280), TensorFHE+ (A100), WarpDrive (A100) in Fig. 11a. The less throughput improvement at higher degree is caused by a higher growth in overall computation from layout invariant NTT at $O(N^{3/2})$ compared with $O(N\log_2 N)$ for others. ***Takeaway:*** This performance gain is attributed to two key optimizations in CROSS: (1) *BAT* reduces redundant compute over zeros and its memory consumption, one type conversion, and halves temporal reduction length. (2) *MAT* eliminates runtime transpose and shuffling by embedding them into computation.

TPUs favor large batch as it reuses common parameters to save off-chip memory access before it overflows on-chip memory, as shown in Fig. 11b. NTT with higher degree benefits less from batching and achieves peak throughput at smaller batch size. This is because it might introduce back-and-forth data access to off-chip HBM, when multi-batch data of high-degree ciphertexts exceed on-chip memory. The optimal batch size for one tensor core of TPUv6e under Set A/B/C/D is 32/16/16/8, giving 7.7×/2.9×/1.5×/1.4× throughput improvement.

Table VII: NTT Throughput (#KNTT/s) Evaluation

Degree	TensorFHE+	WarpDrive	v4-4	v5e-4	v5p-4	v6e-8
$N=2^{12}$	1116	12181	1284	4878	7274	14668
$N=2^{13}$	546	4675	323	1276	1812	3850
$N=2^{14}$	276	2088	75	223	407	793



(a) NTT/s vs TensorFHE+(A100) (b) Impact of Batch Size.

Figure 11: Ablation study: Impact of different hardware and batch size on NTT throughput. Parameters defined in Tab. IV.

C. Evaluating Performance of HE Kernels

We scale TPUs to achieve roughly the same power as other devices in prior works and compare amortized single-batch latency of HE kernels in Tab. VIII, adopting the security parameters that yielded the peak performance reported in original publications of prior works or the corresponding double rescaling version with twice number of 32-bit moduli.

a) CROSS vs SoTAs: After scaling Tensor Cores to match the power envelope of prior systems, CROSS on a single TPUv6e VM achieves significant speedups for HE-Mult/Rotate: 415×/498× over OpenFHE, 1.55×/2.23× over FIDESlib, 1.21×/1.45× over FAB, 2.2×/1.58× over HEAP, and 6×/9.54× over WarpDrive. Compared to Cheddar [17], the SoTA GPU library, we utilize a configuration ($dnum=3$, 65 total moduli) that incurs 1.22× higher memory consumption than the baseline ($dnum=12$, 53 total moduli). Despite this overhead, CROSS achieves speedup performance in HE-Mult (1.1×) and Rotate (1.21×). On average, CROSS/TPUv6e delivers 451×, 7.81×, 1.83×, 1.31×, 1.86×, and 1.15× higher energy efficiency than OpenFHE, WarpDrive, FIDESlib, FAB, HEAP, and Cheddar, respectively. Even against non-public specialized ASICs, CROSS outperforms BASALISC (1.20×) and CraterLake (1.32×) in HE-Add efficiency, while narrowing the HE-Mult performance gap to 3×, 33× and Rotate gap to 2.4×, 33×. These establish CROSS on TPUv6e as SoTA energy-efficiency frontier for HE operators in commodity devices.

b) CROSS for CPU: On the AMD 9950X3D, CROSS delivers up to **5.46×/3.15× speedup** in HE-Mult/Rotate over OpenFHE. The improvement stems from offloading partial computation of redundant zeros and runtime data reordering to offline (compile time). Note that CROSS enables CPU to

Table VIII: Latency and energy efficiency of HE kernels (default $N=2^{16}$). Latency: μs. Values of baseline (in gray) come from their original paper. The TPU configuration consuming similar power as each baseline is highlighted in green. Speedup is ratio of gray to green using the configuration setup of the baseline. Takeaway: CROSS enables TPUv6e to obtain lower amortized single-batch latency and SoTA energy efficiency.

Library	$L, \log_2 q, Dnum$	HE-Add	HE-Mult	Rescale	Rotate
FIDESlib (RTX4090) [5]	30, 59, 3	**51**	**1084**	**156**	**1107**
CROSS (v4-8)		**13.1**	**7336**	**1195**	**4491**
CROSS (v5e-4)		**17.6**	**2201**	**309**	**1418**
CROSS (v5p-8)	60, 28, 3	**6.5**	**1101**	**158**	**993**
CROSS (v6e-4)		**8.2**	**1364**	**187**	**941**
CROSS (v6e-8)		**4.0**	**697**	**95**	**496**
Cheddar (RTX4090) [17]	$48, \leq 31, 12$	**48**	**533**	**68**	**476**
CROSS (v4-8)		**15.5**	**5652**	**969**	**3799**
CROSS (v5e-4)		**13.6**	**1480**	**236**	**1161**
CROSS (v5p-8)	48, 28, 3	**5.6**	**851**	**143**	**769**
CROSS (v6e-4)		**6.5**	**951**	**145**	**787**
CROSS (v6e-8)		**3.5**	**487**	**74**	**393**
FAB (U280) [4]	32, 52, 4	**40**	**1710**	**190**	**1570**
CROSS (v4-8)		**18.6**	**7992**	**1260**	**5253**
CROSS (v5e-4)		**19.0**	**2238**	**344**	**1723**
CROSS (v5p-8)	64, 28, 4	**5.9**	**1190**	**173**	**1098**
CROSS (v6e-4)		**8.8**	**1414**	**194**	**1080**
CROSS (v6e-8)		**6.3**	**709**	**97**	**547**
HEAP (8×U280) [2]	$N=2^{13}, log_2 Q=216$	**1**	**28**	**10**	**25**
CROSS (v4-8)		**18.2**	**197.7**	**84.6**	**241.7**
CROSS (v5e-4)		**2.5**	**34.0**	**5.1**	**40.5**
CROSS (v5p-8)	Set B 8, 28, 3	**7.8**	**48.0**	**17.3**	**77.4**
CROSS (v6e-4)		**3.2**	**20.7**	**17.3**	**31.0**
CROSS (v6e-8)		**6.5**	**12.7**	**11.2**	**15.9**
BASALISC (ASIC) [27]	32, 40, 3	**8**	**312**	**N/A**	**313**
CROSS (v4-8)		**14.9**	**5825**	**919**	**3659**
CROSS (v5e-4)		**14.5**	**1559**	**228**	**1091**
CROSS (v5p-8)	47, 28, 3	**5.7**	**1072**	**137**	**818**
CROSS (v6e-4)		**6.6**	**955**	**135**	**754**
CROSS (v6e-8)		**3.6**	**488**	**67**	**328**
WarpDrive (A100) [24]	34, 28, Not Known	**61**	**4284**	**241**	**5659**
CROSS (v4-8)		**42.8**	**4373**	**724**	**2875**
CROSS (v5e-4)		**14.0**	**1114**	**175**	**908**
CROSS (v5p-8)	36, 28, 3	**11.0**	**656**	**117**	**687**
CROSS (v6e-4)		**10.9**	**714**	**106**	**593**
CROSS (v6e-8)		**5.0**	**358**	**61**	**307**
CraterLake (ASIC) [56]	51, 28, 3	**9**	**35**	**9**	**27**
OpenFHE (AMD 9950X3D)	51, 28, 3	**15390**	**417651**	**22670**	**397798**
CROSS (AMD 9950X3D)		79	132472	27281	72741
CROSS (v4-8)		**15.8**	**5962**	**988**	**3975**
CROSS (v5e-4)	**Default**	**15.0**	**1574**	**242**	**1149**
CROSS (v5p-8)	Set D	**11.0**	**782**	**131**	**853**
CROSS (v6e-4)	51, 28, 3	**6.8**	**1007**	**149**	**798**
CROSS (v6e-8)		**3.5**	**509**	**77**	**414**
Energy Efficiency Improvement Over Publicly Available Devices	vs OpenFHE	**2253×** ↑	**415×** ↑	**152×** ↑	**498×** ↑
	vs FIDESlib	**12.8×** ↑	**1.55×** ↑	**1.64×** ↑	**2.23×** ↑
	vs WarpDrive	**5.61×** ↑	**6.00×** ↑	**2.27×** ↑	**9.54×** ↑
	vs Cheddar	**13.6×** ↑	**1.10×** ↑	**0.92×**	**1.21×** ↑
	vs FAB	**4.55×** ↑	**1.21×** ↑	**0.98×**	**1.45×** ↑
	vs HEAP	**0.15×**	**2.20×** ↑	**0.89×**	**1.58×** ↑
Over Unavailable Devices	vs BASALISC	**1.20×** ↑	**0.33×**	**N/A**	**0.42×**
	vs CraterLake	**1.32×** ↑	**0.03×**	**0.06×**	**0.03×**

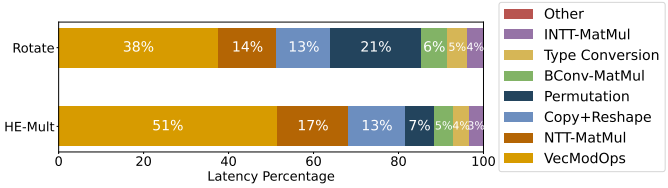


Figure 12: Latency breakdown of HE multiplication and rotation on TPUv6e, under Set D from Tab. VIII.

run faster by switching from $O(N\log_2 N)$ complexity radix-2 cooley-tukey NTT algorithm into $O(N\sqrt{N})$ layout invariant 3-step NTT algorithm, highlighting the need to unleash modern hardware's efficiency by performing workload at coarse granularity of vector or matrix instead of individual elements.

c) Effects of Security Parameters: Increasing (1) the total number of limbs or (2) the digit number (*dnum*), which specifies the number of partitions used for digit decomposition in hybrid key switching [37], could both increase the required computation, leading to longer latency on TPU (Tab. VIII).

d) Latency Breakdown: Fig. 12 provides a runtime breakdown for HE-Mult and Rotate on a single TPUv6e Tensor Core (Set D), identifying key execution bottlenecks.

• **HE-Mult**: Matrix multiplications in NTT/INTT/BConv dominate arithmetic complexity but only contribute 25% of latency. VecModMul instead takes 51%, making HE-Mult VPU-bound. The remaining 13% + 7% is XLA-induced memory relayout to (8,128) tiles to better utilize VReg, not algorithmic cost. BAT adds 4% extra type conversion to change 32-bit data into bytes, as it might trigger layout conversion.

• **Rotate**: The same VPU-bounded bottleneck shows again, dominating 38% latency while MatMuls from NTT, INTT and BConv take only 24% latency, highlighting the efficiency and speed of MXU. The 21% permutation represents worst-case cost, which introduces random gather/scatter when MAT cannot embed a given permutation pattern into computation.

Takeaway: TPUv6e achieves SoTA energy efficiency among commodity devices. Both HE-Mult and Rotate are bounded by vectorized operation with Rotate further bounded by permutation, motivating future work to optimize algorithms to further embed these algorithmic permutation into computation and to convert vectorized operations into low-precision MatMul to be accelerated by high-throughput MXU. Moreover, CROSS's optimizations generalize to any architecture with dedicated matrix or vectorized engine.

D. Evaluating Performance of HE ML Workload

a) MNIST: We evaluate CROSS on a convolution network ($2\times\{$Conv-ReLU-AvgPool$\}\rightarrow$FC$\rightarrow$ReLU$\rightarrow$FC) [67] using a batch size of 64 with $3\times32\times32$ MNIST images [13]. The HE parameters are set to $N=2^{13}$, dnum $=3$, $L=18$, and $\log_2 q=28$ without bootstrapping. Weights are encoded as plaintexts while inputs are encrypted as ciphertexts. On TPUv6e-8, CROSS achieves an amortized inference latency of 270 ms per image, a 10× speedup over Orion [23] with the same 98% accuracy. It's because BAT and MAT optimizations are mathematically lossless transformations that preserve the exact computational results of the modular arithmetic.

b) Logistic Regression: CROSS uses one TPUv6e tensor core to achieve 84 ms for a single iteration of LR, achieving 1.06× throughput per watt than Cheddar (RTX 4090).

Takeaway: We show that TPUv6e as an AI ASIC can be a promising platform for accelerating HE ML workloads. It favors lower precision moduli, giving better energy efficiency for HE ML workloads without bootstrapping.

Table IX: Packed Bootstrapping (v6e-8's Speedup/breakdown)

Work	FIDESlib	Cheddar	CL	v4-8	v5e-4	v5p-8	v6e-8
Latency (ms)	169	31.6	3.91	129.8	59.2	68.3	21.5
v6e's Speedup	**7.9**×	**1.5**×	**0.2**×	**6**×	**2.8**×	**3.2**×	1
v6e-8 Breakdown	Automorphism		VecModMul	(I)NTT		VecModAdd	BConv
Lat. Ratio	35.64%		25.55%	16.87%		15.29%	6.65%

E. Evaluating Packed Bootstrapping

CROSS adopts pack bootstrapping [3] under default Set D from Tab. IV. CROSS on TPUv6e-8 achieves 1.5× throughput (measured by number of bootstraps-per-second) over the SoTA GPU library Cheddar [24], [26], [63] on an NVIDIA RTX 4090 GPU, validating the efficacy of redundancy reduction in BAT and layout reordering elimination in MAT. CROSS enables high compute utilization for HE kernels of the bootstrapping, making TPUv6e the SotA throughput machine for workloads built on HE operators with bootstrapping too.

CROSS on TPUv6e shows 5× bootstrapping throughput gap to HE ASIC, CraterLake [56], when scaling tensor cores to consume roughly same power. Software reasons are:

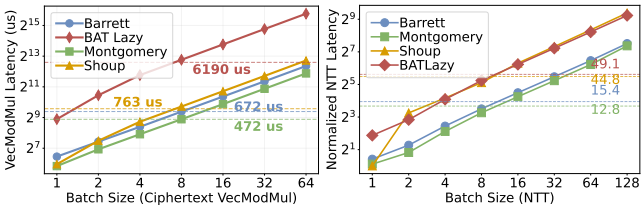
• **Limited Inter-Kernel Optimization**: CROSS does not explore pipelining and fusion between sequential vectorized operators and matrix multiplication of HE kernels of bootstrapping. Consequently, intermediate results are written back to HBM, incurring back-and-forth memory access.

• **Inefficient Permutation in Automorphism**: CROSS's MAT could embed all reordering of NTT but not all permutations in Automorphism into computation. Therefore, automorphism is mapped as random scatter and gather of many degree-length vectors, causing layout transformations. How to embed arbitrary permutation into computation remains an open-question.

F. Ablation Study

1) Impact of Batch Size: As shown in Fig. 11b, increasing batch size improves NTT throughput on TPUs by enabling reuse of shared parameters across ciphertexts. Such parameters include twiddle factors, evaluation keys, and CRT primes for basis conversion. This reuse reduces redundant off-chip memory traffic and increases operational intensity, shifting execution from a memory-bound towards a compute-bound regime. However, excessively large batches exceed on-chip memory capacity, causing contention between input coefficients and parameters and triggering repeated off-chip accesses, which degrades performance. Increasing on-chip memory capacity would give higher benefits at larger batch sizes. Under Set D, we observe that e-class TPUs (e.g., TPUv5e and TPUv6e) achieve peak performance with single-batch HE operators, whereas p-class TPUs (e.g., TPUv4 and TPUv5p) continue to benefit from amortized latency reductions at batch sizes of 2 or 4 because of large off-chip and on-chip memory.

2) Impact of Modular Reduction Algorithm: Since VecModMul and NTT dominate HE compute latency, we evaluate three modular reduction algorithms, including *Barrett* [10], *Montgomery* [49], and *Shoup* [61], across varying batch sizes (Fig. 13). Our results indicate that Montgomery reduction is optimal for TPUv6e for both VecModMul and ModMatMul.



(a) VecModMul w/ diff. mod red. (b) NTT w/ diff. mod red.

Figure 13: Ablation study: Impact of modular reduction (modred) on latency of NTT and VecModMul (Set D).

• **Setup:** We apply the BAT to Barrett and Montgomery reductions to optimize ModMatMul while apply reduction algorithms directly for VecModMul. As Shoup's reduction relies on precompiled parameters that are incompatible with BAT, we implement it using the SoTA GPU high-precision scalar multiplication flow shown in Fig. 7. We also evaluate "BAT lazy reduction", for which we apply BAT to reformulate modular reduction as low-precision matrix multiplication (details in §J). All experiments use a single TPUv6e Tensor Core under security parameter Set D.

• **VecModMul:** Montgomery reduction achieves a 1.42× geomean speedup over Barrett across all batch sizes by reducing runtime computation. While Shoup's algorithm has the lowest theoretical arithmetic complexity, it needs 64-bit multiplication, hence being slower than Montgomery. BAT lazy reduction performs poorly as its reduction dimension is limited to the number of bytes in the data, i.e. $K = 4$ for 32-bit data, resulting in inefficient MXU utilization.

• **NTT:** NTT composes *ModMatMul* and *VecModMul* (Fig. 10). Crucially, the throughput gains from BAT-optimized MatMul further magnify gap between Montgomery and Shoup. The only exception is the single-batch configuration, where the NTT kernel becomes memory-bound on TPUv6e, masking the computational advantages of Montgomery and Barrett.

G. Unveiling Performance Gap to Dedicated FHE ASICs

HE ASIC SoTA: Under HE-Mult / Rotate / bootstrapping as the workload, compared against HE ASIC accelerators CraterLake [56], TPUv6e with CROSS is 33.2×/33.2×/5× slower. The performance gap between CROSS on TPU and FHE ASICs is driven by three factors from hardware side:

- **Hardware-Friendly Moduli:** Dedicated FHE ASICs [56] use fixed moduli (e.g., $2^{32} - v$, with v as a 16-bit value) optimized for hardware. In contrast, CROSS supports arbitrary moduli, which can incur a 2∼3× performance penalty depending on the implementation.
- **Low-Cost All-to-All Shuffling:** Dedicated accelerators feature efficient shuffling mechanisms, such as the layout transpose unit in CraterLake [56] and the all-to-all connected NoC in FAB [4], which enable a butterfly recursive NTT with $O(N \log_2 N)$ complexity. This design choice yields up to a 16× performance advantage over layout invariant 3-step NTT with $O(N\sqrt{N})$ computational complexity when picking polynomial degree $N = 2^{16}$.
- **Extensive Compute and Large On-Chip Memory:** Dedicated FHE ASICs allocate a significant portion of chip area to on-chip memory (e.g., 256 MB, double of TPUv4), which supports larger batch sizes and increased data reuse, thereby enhancing overall performance.

All three factors explain 33× latency gap in Tab. VIII.

VI. Related Work

Previous studies on hardware-accelerated HE follow two main tracks. The first track proposes custom ASIC designs with the sole purpose of accelerating HE workloads. Initial proposals were relatively small [55] but quickly grew to chips requiring hundreds of MB in memory and hundreds of mm^2 in area [41], [56], [59], [73]. While these designs achieve significant performance gains over CPU baselines, they would cost millions of dollars to fabricate and deploy.

This motivates the second track of prior works, which use existing hardware to accelerate HE workloads. Following the path of AI acceleration, these works focus on GPU acceleration [5], [6], [11], [17], [21], [25], [26], [35], [40], [42], [46], [59], [60], [65], [66], [68], [70], [72], [74] as well as employing FPGAs [4], [53], [54], [62] as more configurable commodity hardware. Our work falls into this second category, proposing a novel compilation techniques to better utilize *existing* hardware to accelerate HE workloads at no additional hardware cost. This work establishs the path of adapting AI accelerators for HE workloads to achieve better energy efficiency than SoTA GPUs and FPGAs solutions, enabling the same chip to support both privacy-preserving AI and AI.

VII. Conclusion

This work introduces CROSS, the first compilation framework that enables AI accelerators, such as Google TPUs, to efficiently execute Homomorphic Encryption (HE) operators, achieving superior performance per watt compared to SoTA FPGA and GPU implementations. CROSS establishes new NTT throughput records previously held by GPUs at low polynomial degrees ($N \leq 2^{12}$). More broadly, CROSS proposes compilation techniques to transform applications with statically scheduled modular arithmetic and deterministic permutations into TPU-friendly computational kernels, and hence allows HE workloads to directly inherit the energy efficiency and throughput of modern AI ASICs without any hardware modification. This result positions AI accelerators (e.g., TPUv6e) as a viable and energy-efficient throughput machine for privacy-preserving computation. We leave application-level optimizations and end-to-end benchmarks to future work.

VIII. Acknowledge

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The TPU access of this work is supported by Google TPU Research Cloud. We thank Todd Austin, Mohit Tiwari, Manoj Kumar, Lok Yan, Baiyu Li, Shruthi Gorantala, Bryant Gipson, Ingrid Verbauwhede, Brandon Reagen, Moin Qureshi, Srini Devadas, Mattan Erez, Karthik Garimella, Suvinay Subramanian, Rashimi Agrawal, Scott Hrastar, Simon Langowski, Shannon Egan, Brock Dorion, and anonymous reviewers for insightful feedbacks.

REFERENCES

[1] R. Agarwal and C. Burrus, "Number theoretic transforms to implement fast digital convolution," *Proceedings of the IEEE*, 1975.

[2] R. Agrawal, A. Chandrakasan, and A. Joshi, "Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 756–769.

[3] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023.

[4] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," 2022.

[5] C. Agulló-Domingo, Óscar Vera-López, S. Guzelhan, L. Daksha, A. E. Jerari, K. Shivdikar, R. Agrawal, D. Kaeli, A. Joshi, and J. L. Abellán, "Fideslib: A fully-fledged open-source fhe library for efficient ckks on gpus," 2025. [Online]. Available: https://arxiv.org/abs/2507.04775

[6] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of rns variants of the bfv homomorphic encryption scheme," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 941–956, 2021.

[7] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine *et al.*, "Homomorphic encryption standard," *Protecting privacy through homomorphic encryption*, 2021.

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

[9] A. A. Badawi and Y. Polyakov, "Demystifying bootstrapping in fully homomorphic encryption," Cryptology ePrint Archive, Paper 2023/149, 2023. [Online]. Available: https://eprint.iacr.org/2023/149

[10] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings on Advances in Cryptology—CRYPTO '86*. Berlin, Heidelberg: Springer-Verlag, 1987, p. 311–323.

[11] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, V. Gopal *et al.*, "Intel HEXL (release 1.2)," https://github.com/intel/hexl, Sep. 2021.

[12] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018. [Online]. Available: http://github.com/google/jax

[13] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach, "Low latency privacy preserving inference," 2019. [Online]. Available: https://arxiv.org/abs/1812.10659

[14] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," Cryptology ePrint Archive, Paper 2016/421, 2016, https://eprint.iacr.org/2016/421. [Online]. Available: https://eprint.iacr.org/2016/421

[16] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6993–7010.

[17] W. Choi, J. Kim, and J. H. Ahn, "Cheddar: A swift fully homomorphic encryption library designed for gpu architectures," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2025, p. 35–49. [Online]. Available: https://doi.org/10.1145/3760250.3762223

[18] G. Cloud, "Profile your model on cloud tpu nodes," 2024. [Online]. Available: https://cloud.google.com/tpu/docs/cloud-tpu-tools

[19] N. Corporation. Matrix multiplication background user's guide. NVIDIA. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

[20] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[21] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: Compiler and runtime for homomorphic evaluation of tensor programs," 2018.

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

[23] A. Ebel, K. Garimella, and B. Reagen, "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*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025.

[24] G. Fan, M. Zhang, F. Zheng, S. Fan, T. Zhou, X. Deng, W. Tang, L. Kong, Y. Song, and S. Yan, "Warpdrive: Gpu-based fully homomorphic encryption acceleration leveraging tensor and cuda cores," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 1187–1200.

[25] G. Fan, F. Zheng, L. Wan, L. Gao, Y. Zhao, J. Dong, Y. Song, Y. Wang, and J. Lin, "Towards faster fully homomorphic encryption implementation with integer and floating-point computing power of gpus," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 798–808.

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

[27] R. Geelen, M. V. Beirendonck, H. V. L. Pereira, B. Huffman, T. McAuley, B. Selfridge, D. Wagner, G. Dimou, I. Verbauwhede, F. Vercauteren, and D. W. Archer, "BASALISC: Programmable hardware accelerator for BGV fully homomorphic encryption," Cryptology ePrint Archive, Paper 2022/657, 2022. [Online]. Available: https://eprint.iacr.org/2022/657

[28] Google, "Google Cloud TPU," 2024. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[29] Google, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," 2025. [Online]. Available: https://arxiv.org/abs/2507.06261

[30] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33019466

[31] S. Jayashankar, E. Chen, T. Tang, W. Zheng, and D. Skarlatos, "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*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 133–150. [Online]. Available: https://doi.org/10.1145/3669940.3707260

[32] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[33] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," 2023.

[34] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. A. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, pp. 67–78, 2020. [Online]. Available: https://doi.org/10.1145/3360307

[35] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[36] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," 2018. [Online]. Available: https://arxiv.org/abs/1801.05507

[37] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," Cryptology ePrint Archive, Paper 2021/204, 2021. [Online]. Available: https://eprint.iacr.org/2021/204

[38] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589053

[39] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.

[40] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2020, p. 264–275. [Online]. Available: http://dx.doi.org/10.1109/IISWC50251.2020.00033

[41] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 711–725. [Online]. Available: https://doi.org/10.1145/3470496.3527415

[42] W.-K. Lee, S. Akleylek, D. C.-K. Wong, W.-S. Yap, B.-M. Goi, and S.-O. Hwang, "Parallel implementation of nussbaumer algorithm and number theoretic transform on a gpu platform: application to qtesla," *J. Supercomput.*, vol. 77, no. 4, p. 3289–3314, Apr. 2021. [Online]. Available: https://doi.org/10.1007/s11227-020-03392-x

[43] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "Error-latency-aware scale management for fully homomorphic encryption," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[44] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "Performance-aware scale analysis with reserve for homomorphic encryption," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024.

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

[46] Q. Li and R. Zong, "Cat: A gpu-accelerated fhe framework with its application to high-precision private dataset query," 2025. [Online]. Available: https://arxiv.org/abs/2503.22227

[47] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[48] V. Lyubashevsky, "Lattice signatures without trapdoors," in *Advances in Cryptology – EUROCRYPT 2012*, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 738–755.

[49] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[50] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2020, pp. 64–70.

[51] OpenAI, "Gpt-4o system card," 2024. [Online]. Available: https://arxiv.org/abs/2410.21276

[52] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," 2023.

[53] X. Ren, Z. Chen, Z. Gu, Y. Lu, R. Zhong, W.-J. Lu, J. Zhang, Y. Zhang, H. Wu, X. Zheng, H. Liu, T. Chu, C. Hong, C. Wei, D. Niu, and Y. Xie, "Cham: A customized homomorphic encryption accelerator for fast matrix-vector product," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[54] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1295–1309. [Online]. Available: https://doi.org/10.1145/3373376.3378523

[55] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252. [Online]. Available: https://doi.org/10.1145/3466752.3480070

[56] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: A hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022.

[57] S. S. Sengar, A. B. Hasan, S. Kumar, and F. Carroll, "Generative artificial intelligence: A systematic review and applications," 2024. [Online]. Available: https://arxiv.org/abs/2405.11029

[58] T. Shimada and M. Ikeda, "High-throughput polynomial multiplier architecture for lattice-based cryptography," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[59] K. Shivdikar, Y. Bao, R. Agrawal, M. Shen, G. Jonatan, E. Mora, A. Ingare, N. Livesay, J. L. Abellán, J. Kim *et al.*, "Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption," *arXiv preprint arXiv:2309.11001*, 2023.

[60] K. Shivdikar, G. Jonatan, E. Mora, N. Livesay, R. Agrawal, A. Joshi, J. Abellan, J. Kim, and D. Kaeli, "Accelerating polynomial multiplication for homomorphic encryption on gpus," 2022. [Online]. Available: https://arxiv.org/abs/2209.01290

[61] V. Shoup *et al.*, "Ntl: A library for doing number theory," 2001.

[62] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[63] Y. Sun, S. Fan, Z. Yin, X. Song, X. Hu, Z. Du, Q. Guo, W. Xu, R. Hou, D. Meng, S. Bian, and M. Zhan, "Tensorfhe+: Fully homomorphic encryption acceleration based on linear algebra," *IEEE Transactions on Computers*, pp. 1–14, 2025.

[64] M. van der Hagen and B. Lucia, "Client-optimized algorithms and acceleration for encrypted compute offloading," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022.

[65] Z. Wang, H. He, L. Zhao, P. Li, Z. Li, D. Meng, and R. Hou, "Chameleon: An efficient fhe scheme switching acceleration on gpus," 2024. [Online]. Available: https://arxiv.org/abs/2410.05934

[66] Z. Wang, P. Li, R. Hou, Z. Li, J. Cao, X. Wang, and D. Meng, "Hebooster: An efficient polynomial arithmetic acceleration on gpus for fully homomorphic encryption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1067–1081, 2023.

[67] WISE-HE. Wise. GitHub repository; main branch; commit f0689fd ("init"). [Online]. Available: https://github.com/WISE-HE/WISE

[68] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 5, p. 4895–4906, Sep. 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2024.3363900

[69] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 870–881.

[70] Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky, "Accelerating encrypted computing on intel gpus," 2021. [Online]. Available: https://arxiv.org/abs/2109.14704

[71] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "Sok: Fully homomorphic encryption accelerators," *ACM Comput. Surv.*, vol. 56, no. 12, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3676955

[72] J. Zhao, H. Yang, M. Hao, W. Zhang, H. He, and D. Wang, "Hengine: A high performance optimization framework on a gpu for homomorphic encryption," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 2, Jul. 2025. [Online]. Available: https://doi.org/10.1145/3732942

[73] M. Zhou, Y. Nam, X. Wang, Y. Lee, C. Wilkerson, R. Kumar, S. Taneja, S. Mathew, R. Cammarota, and T. Rosing, "Ufc: A unified accelerator for fully homomorphic encryption," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 352–365.

[74] Y. Zhu, X. Wang, L. Ju, and S. Guo, "Fxhenn: Fpga-based acceleration framework for homomorphic encrypted cnn inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 896–907.

APPENDIX

A. Abstract

We provide scripts to reproduce latency of BAT (Tab. V) and BConv (Tab. VI), throughput of NTT (Tab. VII, Fig. 11a and Fig. 11b), latency of HE operators (Tab. VIII), performance of different modular reduction algorithms (Fig. 13a and Fig. 13b), latency profiling (Fig. 12), and packed bootstrapping estimation (Tab. IX). We provide individual script to run each experiment and obtain the final results. From §F, we provide necessary background, experimental setup, and additional results for precise results reproduction and easily adopting CROSS's contributions.

B. Artifact check-list (meta-information)

- **Run-time environment:** Python 3.13, jax[tpu]
- **Hardware:** TPUv4, TPUv5e, TPUv5p, TPUv6e
- **Experiments:** Critical experiments (Tab. V, Tab. VI, Fig. 11a, Tab. VIII), Optional Experiment (Tab. IX, Fig. 13a, Fig. 13b, Tab. VII, Fig. 12).
- **Publicly available?:** https://github.com/EfficientPPML/CROSS
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.18077699

C. Description and Installation

The CROSS framework runs on any device supporting JAX. For reproducing the performance on TPU, the access of TPUv4,v5e,v5p,v6e is required. And it runs on Python 3.13 with following packages. We note that if running on GPU, the data type for 8-bit convolution should be changed from uint8 into int16 as GPU does not support uint8 based convolution.

```
pip install -U "jax[tpu]", xprof, absl-py,
    pandas, gmpy2
```

D. Experiment workflow

Contact authors to obtain the access of TPUs, and then run each provided script to get results for individual experiments. The entire repository is organized by absltest, and profiled by XProf with the compiled kernel latency being printed out in the terminal and written out as a csv.

E. Evaluation and expected results

Values in the tables and figures are expected results, and the difference of profiled results should have $\leq \pm 5\%$ difference compared to the provided results in the paper.

F. Compute Pattern of Bottleneck Kernels in SoTA HE Library

HE workloads fundamentally boil down to the scheduled invocation of essential HE operators, including HE Multiplication (HE-Mult), Relinearization (Relin.), Rotate, and Rescale. In other words, the performance of these operators directly determines the overall serving latency of HE workloads. To identify performance bottlenecks, we profiled SoTA HE algorithms used by HE FPGA [4] and ASIC [38] on an AMD Ryzen 9 5950X CPU with AVX support using OpenFHE library. The latency breakdown, presented in Fig. 14, reveals NTT, and its inverse (INTT), BConv, and Vectorized Modular Multiplication (`VecModMul`) and Addition (`VecModAdd`) are the five most time-consuming HE kernels in both CKKS and BFV schemes.

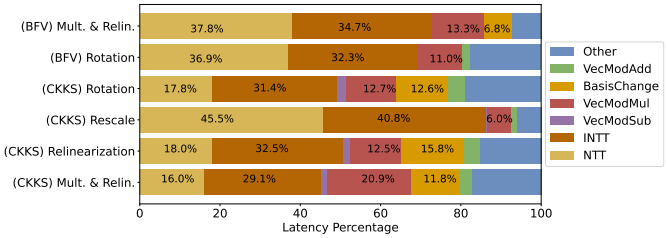


Figure 14: Latency profiling of HE operators using OpenFHE, picking representative values for the parameters in Tab. I. Vectorized Modular Multiplication (`VecModMul`), Addition (`VecModAdd`) and Subtraction (`VecModSub`) only count for latency not belonging to (I)NTT/BConv.

*1) Radix-2 Cooley-Tukey NTT algorithm (**Butterfly NTT**):* The NTT converts polynomial representations from the coefficient domain to the evaluation domain, where polynomial multiplication simplifies to element-wise (vectorized) coefficient multiplication. The NTT and INTT are computationally intensive, accounting for approximately 45.1% to 86.3% of the overall latency in various HE operators.

The detailed algorithm is provided in Algorithm 3. In general, an N-point NTT consists of $\log_2(N)$ stages. Compute-wise, each stage comprises $\frac{N}{2}$ vectorized modular multiplications, additions, and subtractions, denoted as $\frac{N}{2}$-`VecModMul`, $\frac{N}{2}$-`VecModAdd`, and $\frac{N}{2}$-`VecModSub`, respectively; Memory-wise, each stage also requires bit-complement shuffling [20].

Considering 8-point NTT depicted in Fig. 15a as an example, in stage 1, we perform a 4-element vectorized modular multiplication between $[a_4, a_5, a_6, a_7]$ and $[\omega, \omega^2, \omega^3, \omega^4]$:

$$[\tilde{a}_4, \tilde{a}_5, \tilde{a}_6, \tilde{a}_7] = [a_4 \cdot \omega,\ a_5 \cdot \omega^2,\ a_6 \cdot \omega^3,\ a_7 \cdot \omega^4] \mod q$$

Then $vec = [a_0, a_1, a_2, a_3, \tilde{a}_4, \tilde{a}_5, \tilde{a}_6, \tilde{a}_7]$ is being reduced with the bit complement shuffled result of itself $vec = vec \pm \text{bit_complement_shuffle}(vec)$ using vectorized modular addition and subtraction to produce:

$$[b_0, b_1, b_2, b_3] = [a_0 + \tilde{a}_4,\ a_1 + \tilde{a}_5,\ a_2 + \tilde{a}_6,\ a_3 + \tilde{a}_7] \mod q$$

$$[b_4, b_5, b_6, b_7] = [a_0 - \tilde{a}_4,\ a_1 - \tilde{a}_5,\ a_2 - \tilde{a}_6,\ a_3 - \tilde{a}_7] \mod q$$

The bit-complement shuffling in the first stage has a group size of 8 elements [20], as illustrated by the permutation arrows in Fig. 15a. Subsequent stages perform bit-complement permutation with the group sizes decreasing by 2 per stage.

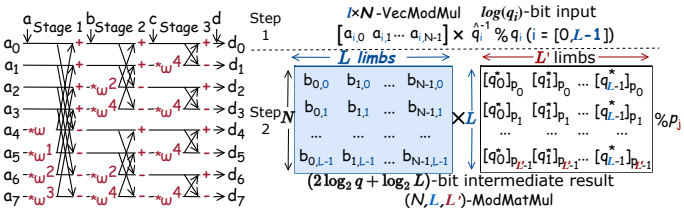
On TPUv4, radix-2 cooley-tukey NTT following its algorithmic processing order runs $\sim 30\times$ slower than MAT-based NTT, even though radix-2 cooley-tukey NTT has lower computational complexity of $O(NlogN)$ than $O(N^{3/2})$ of MAT-based NTT, as shown in Tab. X. This is because (1) heavy reordering and (2) zero MXU utilization.

*2) Basis Conversion (**BConv**):* BConv is used to scale up or down polynomials in HE-Mult and Rotate. For instance, the

Algorithm 3 Radix-2 Cooley-Tukey NTT [1]

Require: Coefficients of input polynomial $P_m = (a_0, a_1, \cdots, a_{N-1})$, N-th root of unity ω, total degree N.
Ensure: $NTT(P_m) = (b_0, b_1, \cdots, b_{N-1})$

1: Initializes $b_i = a_i, i \in [0, N-1]$; $\omega' = \omega$
2: **for**($n = N$, $n > 1$, $n = n \gg 1$) ▷ Stage index
3: $\omega_n = \omega^{\frac{N}{n}} \mod q$
4: **for**($i = 0$, $i < n$, $i = i+n$)
5: **for**($j = 0$, $j < n$, j++)
6: $b[k+j] = (b[k+j] + \omega' b[k+j+n/2]) \mod q$
7: $b[k+j+n/2] = (b[k+j] - \omega' b[k+j+n/2]) \mod q$
8: $\omega' = \omega \cdot \omega_n \mod q$



(a) 8-input butterfly NTT (b) Basis Change (BConv)

Figure 15: Illustration of computation patterns of butterfly NTT and basis change, superscript represents exponent.

Table X: Comparison (Radix-2 CT-NTT vs. MAT-based NTT on TPUv4). Latency of 128-batch NTTs (unit: μs, $N = R \times C$).

Degree (N)	R	C	Radix-2 CT-NTT	MAT NTT	Speedup
2^{12}	128	64	2420	91.8	**26.39**×
2^{13}	128	64	4999	165.4	**30.23**×
2^{14}	128	128	10530	355.5	**29.5**×
2^{15}	256	128	22228	812.3	**27.24**×
2^{16}	256	128	46996	1844.8	**25.47**×

basis conversion from $\mathscr{B}_1$ (moduli: q_i, $i \in [0, L)$) to $\mathscr{B}_2$ (target moduli: p_j, $j \in [0, L')$) [14] is expressed as:

$$Conv_{\mathscr{B}_1 \to \mathscr{B}_2}(\mathbf{a}) = (\Sigma_{i=0}^{L-1} [a_{n,i} \cdot \hat{q}_i^{-1}]_{q_i} \cdot [q_i^*]_{p_j} \mod p_j)_{0 \le j < L', 0 \le n < N}$$

where $\hat{q}_i^{-1}$ and $[q_i^*]_{p_j}$ could be generated offline and get loaded into the on-chip memory as static parameters during runtime.

The above equation is broken into two steps (Fig. 15b).

- Step 1: $b_{n,i} = [a_{n,i} \cdot \hat{q}_i^{-1}]_{q_i}, 0 \le i < L, 0 \le n < N$ invokes L independent instances of N-length Vectorized Modular Multiplication, noted as $L \times N$-`VecModMul` for simplicity.
- Step 2: $c_{n,j} = \Sigma_{i=0}^{L-1} b_{n,i} \cdot [q_i^*]_{p_j} \ mod \ p_j, 0 \le j < L', 0 \le n < N$. It invokes one $M_{N \times L'} = M_{N \times L} \cdot M_{L \times L'}$ Modular Matrix Multiplication, noted as (N, L, L')-`MatModMul`

G. Final Barrett Modular Reduction in CROSS

CROSS adopts lazy reduction to allow data to be temporarily over-precision during a chain of compute and defer the final modular reduction to the final result. Specifically, the Montgomery Reduction in Alg. 1 cannot be used as final modular reduction because its output range is $[0, 2 \cdot q_i)$ instead of the desired range $[0, q_i)$. In CROSS, the final modular reduction is achieved through Barrett modular reduction [10].

Specifically, Barrett converts modular multiplication into two multiplications, one shifted multiplication, up-to two subtractions listed in Alg. 4 with precision listed in comments.

Algorithm 4 Modular Multiplication and Barrett Reduction

Require: $a, b, m \in \mathbb{Z}_q$, $s = 2\lceil \log_2 q \rceil$, $m = \lfloor 2^s / q \rfloor$.
Ensure: $z = a \cdot b \mod q$

1: $z \leftarrow a \cdot b$ ▷ $\log_2 q \times \log_2 q \to 2\log_2 q$
2: $t \leftarrow (z \cdot m) \gg s$ ▷ $2\log_2 q \times \log_2 q \gg 2\log_2 q \to \log_2 q$
3: $z \leftarrow z - (t \cdot q)$ ▷ $2\log_2 q - \log_2 q \times \log_2 q \to 2\log_2 q$
4: **if** $z \ge q$
5: $z \leftarrow z - q$

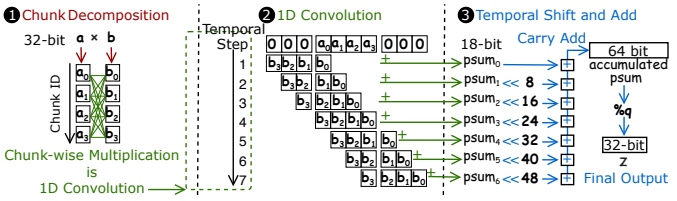


Figure 16: CROSS maps *high-precision scalar multiplication* into 1D convolution and temporal shifted accumulation when input operands are not known a priori.

H. Fall-back Algorithm for unknown parameters of Fig. 7

When all input operands in 32-bit integer arithmetic are not preknown, such as multiplying two input values, BAT is no longer applicable. In this case, CROSS falls back to schedule low-precision chunk-wise multiplications as 1D convolution, as shown in Fig. 16.

Taking mapping 32-bit coefficient multiplication to $bp = 8$-bit compute as an example, as shown by ❶ in Fig. 16, CROSS segments each coefficient into four uint8 chunks and then maps chunk-wise multiplication as 1D convolution (❷). Specifically, CROSS reads uint32 from local register, views it as a vector of four uint8 chunks, and pads $\frac{32}{bp} - 1 = 3$ zeros on both sides. Then, convolving the padded a with chunk-decomposed b over seven temporal cycles yields seven partial sums ($psum^k, k \in [0, 6]$), each at most $2bp + \log_2 \frac{32}{bp} = 16 + 2 = 18$ bits[3]. These partial sums are shifted and accumulated (❸) to 64-bit final result $psum$, which is stored in eight uint8 registers. $psum$ is finally modular reduced to 32 bits via Barrett Reduction (Alg. 4). This achieves the same efficiency as sparse matrix multiplication used in TensorFHE(+) [26]. This algorithm is not used for HE operators, and we detail it here to ensure the general applicability of CROSS.

I. Applying BAT to Preknown Parameters

We provide Alg. 5 to formalize the BAT-enhanced data flow depicted in Fig. 7. BAT is applied to NTT twiddle factors and basis conversion coefficients to maximize throughput on the TPU's MXU. We exclude evaluation keys from BAT because their small reduction size ($dnum \times$ bytes ≈ 12) leads to severe MXU under-utilization. While inefficient for the TPU's large systolic arrays, such a transformation could benefit devices with smaller fine-grained matrix engine such as GPUs.

[3]Each chunk-wise multiplication generates 16 bits. A reduction of $\frac{32}{bp} = 4$ chunks needs at most 2 extra bits to avoid precision overflow.

Algorithm 5 High-Precision Scalar Multiplication using BAT.

Require: modulus q; $a,b,z \in \mathbb{Z}_q$ have $\log_2 q$ bits; hardware supports bp-bit arithmetic; $K \leftarrow \lceil \frac{\log_2 q}{bp} \rceil$, indicating number of bytes.

CONSTRUCTTOEPLITZ$([a_k]_{0\le k<K}) \to X$ ▷ ❶ in Fig. 7.
1: $X \leftarrow \text{zeros}(2K-1, K)$
2: **for** $j = 0$ **to** $K-1$:
3: **for** $i = 0$ **to** $K-1$:
4: $X[i+j, j] \leftarrow a_j$
5: **Return** X

BAT$(X, bp, q) \to X$ ▷ ❸ in Fig. 7.
6: **for** $i = 0$ **to** $K-2$: ▷ Iterate row in bottom block
7: **for** $j = 0$ **to** $K-2-i$:
8: $basis \leftarrow (K+i) \cdot bp$
9: $proj \leftarrow (X[i+j+1, j] \ll basis) \bmod q$
10: $[r_k]_{0\le k<3} \leftarrow$ CHUNKDECOMPOSE$(proj)$
11: $X[k, K-1-j] \leftarrow X[k, K-1-j] + r_k,\ 0 \le k < 3$
12: **Return** X

CARRYPROPAGATION$(X,$ bp$) \to X$
▷ Ensure all values in $X \le 2^{bp}-1$.
13: **for** $j = 0$ **to** $K-1$: ▷ Iterate all columns
14: **for** $k = 0$ **to** (#Rows of X)-2:
15: **if** $X[k, j] > 2^{bp}-1$:
16: $carry \leftarrow \lfloor X[k, j] / 2^{bp} \rfloor$
17: $X[k, j] \leftarrow X[k, j] \bmod 2^{bp}$
18: $X[k+1, j] \leftarrow X[k+1, j] + carry$
19: **Return** X

OFFLINECOMPILE$(X, K, bp, q) \to X_{dense}$ ▷ ❹ in Fig. 7.
20: $[a_k]_{0\le k<K} \leftarrow$ CHUNKDECOMPOSE(a)
21: $X \leftarrow$ CONSTRUCTTOEPLITZ$([a_k]_{0\le k<K})$ ▷ ❶ in Fig. 7.
22: **while** (*not all* $X \le 2^{bp}-1$ **or** $X[K:,:] \ne 0$):
23: $X \leftarrow$ CARRYPROPAGATION(X)
24: **if** $X[K:,:] \ne 0$ ▷ Bottom block contains non-zero.
25: $X \leftarrow$ BAT(X)
26: **Return** $X[0:K, 0:K]$ ▷ All values cast to uint8.

MAIN-HPSCALARMULT$(a, b) \to z$
27: $\left[\hat{a}_{i,k}\right]_{0\le i,k<K} \leftarrow$ OFFLINECOMPILE(a) ▷ ❹ in Fig. 7.
28: $[b_k]_{0\le k<K} \leftarrow$ CHUNKDECOMPOSE(b)
29: $[c_k] = \left[\hat{a}_{i,k}\right] \times [b_k],\ 0 \le i,k < K$
30: **for** $k = 0$ **to** $K-1$:
31: $z \mathrel{+}= c_k \ll (bp * k)$ ▷ ❺ in Fig. 7.
32: **Return** z

J. BAT for Modular Reduction

The primary objective of the modular reduction after performing 32-bit multiplication into 64- or 48-bit partial sum in Fig. 7 is to compress the 64- or 48-bit partial sum into 32-bit value range required by subsequent pipeline stages, i.e., the final value needs to fit in 32 bits but its actual value could be larger than q. Therefore, instead of full modular reduction, we implement a lazy partial reduction to bring 64- or 48-bit partial sum to fit 32 bits (but might larger than q). This allows us to only target the "overflow" bits beyond the 32-bit boundary and apply BAT to it, making it a low-precision MatMul.

This BAT-based formulation enables offloading the modular reduction to the matrix engine, replacing a sequence of vectorized shifts and additions. However, we omit this optimization in our final TPU implementation, because the 4×4 ($K=4$) reduction dimension is insufficient to saturate the TPU's 128×128 Matrix Unit (MXU). The resulting systolic array under-utilization makes standard vector instructions more efficient. Conversely, this technique is well-suited for architectures with finer-grained tensor engines (e.g., GPUs), where smaller tile granularities can effectively exploit such small matrix multiplications to achieve higher throughput.

Specifically, the detailed math behind BAT lazy reduction is listed below, assuming multiplication of a and b produces a 64-bit *psum*, which is decomposed into $2K$=8 bytes c_j, $j \in [0, 8)$.

$$
\begin{aligned}
z = psum \bmod q &= \sum_{j=0}^{j=2K-1} \Big(\underbrace{c_j}_{\text{one byte}} \times 2^{8j} \Big) \bmod q \\
&= \left(\underbrace{\sum_{j=K}^{j=2K-1} (c_j \times 2^{8j})}_{\text{High 32 bits, need reduction}} + \underbrace{\sum_{j=0}^{j=K-1} (c_j \times 2^{8j})}_{\text{Low 32 bits noted as } low} \right) \bmod q \\
&= \left(\sum_{j=K}^{j=2K-1} \Big(c_j \times \underbrace{(2^{8j} \bmod q)}_{\text{(BAT) precomputed as } LC_j}\Big) + low \right) \bmod q \\
&= \left(\sum_{j=0}^{j=K-1} \Big(c_{j+K} \times \underbrace{LC_j}_{K \text{ Bytes each}}\Big) + low \right) \bmod q \\
&= \left(\sum_{j=0}^{j=K-1} \Big(c_{j+K} \times \sum_{k=0}^{k=K-1} \Big(\underbrace{LC_{j,k}}_{\text{one byte each}} \times 2^{8k} \Big)\Big) + low \right) \bmod q \\
&= \left(\sum_{k=0}^{k=K-1} \left(\sum_{j=0}^{j=K-1} c_{j+K} \times LC_{j,k} \right) \times 2^{8k} + low \right) \bmod q \\
&= \left(\sum_{k=0}^{K-1} \begin{bmatrix} c_K \\ c_{K+1} \\ \vdots \\ c_{2K-1} \end{bmatrix} \times \underbrace{\begin{bmatrix} LC_{0,0} & \cdots & LC_{0,K-1} \\ LC_{1,0} & \cdots & LC_{1,K-1} \\ \vdots & \ddots & \vdots \\ LC_{K-1,0} & \cdots & LC_{K-1,K-1} \end{bmatrix}}_{K\times K \text{ 8-bit matrix}} \cdot \begin{bmatrix} 2^0 \\ 2^8 \\ \vdots \\ \vdots \end{bmatrix} + low \right) \bmod q
\end{aligned}
$$