MetaKernel: Enabling Efficient Encrypted Neural Network Inference through Unified MVM and Convolution

PENG YUAN, Ant Group, China
YAN LIU, Ant Group, China
JIANXIN LAI, Ant Group, China
LONG LI, Ant Group, China
TIANXIANG SUI, Ant Group, China
LINJIE XIAO, Ant Group, China
XIAOJING ZHANG, Ant Group, China
QING ZHU, Ant Group, China
JINGLING XUE, UNSW, Australia and Ant Group, China

Practical encrypted neural network inference under the CKKS fully homomorphic encryption (FHE) scheme relies heavily on accelerating two key kernel operations: Matrix-Vector Multiplication (MVM) and Convolution (Conv). However, existing solutions—such as expert-tuned libraries and domain-specific languages—are designed in an ad hoc manner, leading to significant inefficiencies caused by excessive rotations.

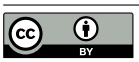
We introduce MKR, a novel composition-based compiler approach that optimizes MVM and Conv kernel operations for DNN models under CKKS within a unified framework. MKR decomposes each kernel into composable units, called *MetaKernels*, to enhance SIMD parallelism within ciphertexts (via horizontal batching) and computational parallelism across them (via vertical batching). Our approach tackles previously unaddressed challenges, including reducing rotation overhead through a rotation-aware cost model for data packing, while also ensuring high slot utilization, uniform handling of inputs with arbitrary sizes, and compatibility with the output tensor layout. Implemented in a production-quality FHE compiler, MKR achieves inference time speedups of 10.08×–185.60× for individual MVM and Conv kernels and 1.75×–11.84× for end-to-end inference compared to a state-of-the-art FHE compiler. Moreover, MKR enables homomorphic execution of large DNN models, where prior methods fail, significantly advancing the practicality of FHE compilers.

CCS Concepts: • **Software and its engineering → Compilers**; • **Security and privacy → Cryptography**;

Additional Key Words and Phrases: FHE, CKKS, FHE Compilers, MetaKernel

ACM Reference Format:
Peng Yuan, Yan Liu, JianXin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Xiaojing Zhang, Qing Zhu, and Jingling Xue. 2025. MetaKernel: Enabling Efficient Encrypted Neural Network Inference through Unified MVM and Convolution. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 317 (October 2025), 28 pages. https://doi.org/10.1145/3763095

Authors' Contact Information: Peng Yuan, Ant Group, Beijing, China, yp398707@antgroup.com; Yan Liu, Ant Group, Shanghai, China, ly409648@antgroup.com; JianXin Lai, Ant Group, Shanghai, China, laijianxin.ljx@antgroup.com; Long Li, Ant Group, Shanghai, China, ll398708@antgroup.com; Tianxiang Sui, Ant Group, Shanghai, China, suitianxiang.stx@antgroup.com; Linjie Xiao, Ant Group, Shenzhen, China, xiaolinjie.xlj@antgroup.com; Xiaojing Zhang, Ant Group, Shanghai, China, zxj398711@antgroup.com; Qing Zhu, Ant Group, Shanghai, China, zq398709@antgroup.com; Jingling Xue, UNSW, Sydney, Australia and Ant Group, Shanghai, China, j.xue@unsw.edu.au.



ACM 2475-1421/2025/10-ART317
https://doi.org/10.1145/3763095

1 Introduction

Problem Statement. Fully Homomorphic Encryption (FHE) [Gentry 2009] allows computation on encrypted data, enabling secure task offloading to untrusted cloud servers. The CKKS scheme [Cheon et al. 2017] is preferred in privacy-preserving machine learning (PPML) [Lee et al. 2022a] due to its support for SIMD parallelism (shared with BGV [Brakerski et al. 2011] and BFV [Fan and Vercauteren 2012]) and unique fixed-point arithmetic capabilities. In practical encrypted inference using DNNs, efficient implementation of two key kernels—Matrix-Vector Multiplication (MVM) and Convolution (Conv)—is crucial. These kernels involve interacting plaintext weights with encrypted (feature) vectors or images, utilizing element-wise ciphertext-plaintext additions and multiplications as well as ciphertext rotations. Each ciphertext and plaintext can hold $\frac{N}{2}$ elements (where N is the polynomial ring degree [Cheon et al. 2017]), necessitating advanced data layout transformations. Given that rotations are substantially more expensive than other operations [Liu et al. 2025; Zhang et al. 2025], minimizing their usage is vital for optimizing MVM and Conv performance.

In this paper, we present a compiler approach to optimize matrix-vector multiplication (MVM) and convolution (Conv) under the CKKS scheme for single-CPU execution. The goal is to minimize costly rotations while maximizing ciphertext slot utilization ($\mathcal{S}$ denotes $\frac{N}{2}$ slots). Efficiently achieving this on a single CPU remains challenging; multi-core extensions are possible but out of scope.

Challenges. Ensuring security requirements [Albrecht et al. 2019; Bossuat et al. 2024] necessitates large slot sizes (typically 32K or 64K in PPML applications, supported by FHE accelerators like F1 [Samardzic et al. 2021] and CraterLake [Samardzic et al. 2022]). Combined with the limited operations in FHE schemes, including CKKS [Cheon et al. 2017], this creates several interrelated challenges for optimizing MVM and Conv kernels on CPUs:

- **C1. Controlled Rotations.** Minimizing costly ciphertext rotations, which arise from the need to align plaintext weights with encrypted vectors in MVM and images in Conv, is a complex task.
- **C2. Data Packing.** The challenge lies in maximizing slot utilization while managing the trade-off between SIMD parallelism and increased rotation costs, which complicates optimization.
- **C3. Cost Model.** Developing a cost model to systematically balance rotation overhead with slot utilization is challenging, especially considering the need for rotation-aware data packing.
- **C4. Uniform Handling of Varying Input Sizes.** DNN inputs vary in size, often exceeding ciphertext capacity and requiring partitioning. A uniform handling of both small and large inputs is required to efficiently map tensors without unnecessarily increasing multiplicative depth.
- **C5. Output Tensor Layout Compatibility.** Ensuring that output tensor layouts match their unencrypted counterparts—without incurring extra rotations in subsequent layers—while achieving high-performance MVM and Conv is essential for seamless integration and efficiency.
- **C6. Unified Approach for MVM and Conv.** Developing a unified framework to optimize both MVM and Conv using a shared abstraction is a challenging and unexplored task.

Table 1. Comparison of MKR with FHE-MP-CNN and Fhelipe in addressing Challenges **C1** - **C6**.

Approach	C1	C2		C3	C4		C5		C6
		MVM	Conv		MVM	Conv	MVM	Conv	
FHE-MP-CNN	✗	Diagonal	Full Replication	✗	✓	✗	✓	✗	✗
Fhelipe	✗	Full Replication	Full Replication	✗	✓	✓	✗	✗	✗
MKR	✓	Rotation-Aware	Rotation-Aware	✓	✓	✓	✓	✓	✓

Prior Work. Research on optimizing MVM and Conv for homomorphic execution is still in its early stages, focusing mainly on enabling these operations through data packing rather than improving them by reducing rotation overhead for better inference efficiency. Existing work falls into two categories: expert-designed libraries, such as CryptoNets [Gilad-Bachrach et al. 2016],

nGraph-HE2 [Boemer et al. 2019], and FHE-MP-CNN [Lee et al. 2022b]—with FHE-MP-CNN as the benchmark—and DSL compilers, including HECO [Viand et al. 2022], CHET [Dathathri et al. 2019], EVA [Dathathri et al. 2020], and Fhelipe [Krastev et al. 2024], with Fhelipe as the leading example. As shown in Table 1, both FHE-MP-CNN and Fhelipe overlook most of the six outlined challenges and inadequately address the rest, leading to significant inference inefficiencies for MVM and Conv.

FHE-MP-CNN provides expert-tuned MVM and Conv implementations. For MVM, it packs each diagonal of the plaintext weight matrix, neglecting slot utilization. For Conv, it fully replicates the encrypted image, achieving 100% slot utilization but producing non-contiguous output channels. FHE-MP-CNN supports only small models like ResNet-20 on CIFAR ($3 \times 32 \times 32$), where each image fits in a single ciphertext, making it unsuitable for larger inputs like ImageNet ($3 \times 224 \times 224$).

Fhelipe [Krastev et al. 2024], a DSL compiler, automates data packing and matches FHE-MP-CNN's performance. For both MVM and Conv, it handles arbitrary input sizes by partitioning them to fit ciphertext slots, applying power-of-2 padding when needed. Fhelipe maximizes slot utilization through full data replication but leaves outputs misaligned, requiring additional rotations for subsequent layers. By focusing on data packing without considering rotation overhead, Fhelipe performs comparably to FHE-MP-CNN but remains suboptimal for MVM and Conv.

Recently, Orion [Ebel et al. 2025] was introduced as an FHE compiler framework for translating DNNs written in PyTorch into FHE programs, supporting ImageNet models such as `ResNet-34` and `ResNet-50`, as well as high-resolution FHE object detection with `YOLO-v1`. Cinnamon [Jayashankar et al. 2025] focuses on system-level techniques for scale-out accelerator design and parallel programming in FHE compilers for models like BERT, achieving significant speedups over CPU execution.

This Work. In this paper, we introduce MKR, a novel composition-based compiler approach for optimizing MVM and Conv under CKKS in a unified framework. Guided by a rotation-aware cost model, MKR minimizes rotation overhead while maintaining high slot utilization. Our approach is based on two key insights: (1) high slot utilization can degrade performance due to excessive rotation overhead from large data replication; and (2) MVM and Conv share similar computation patterns, enabling unified optimization through divide-and-conquer decomposition.

MKR addresses all six challenges outlined in Table 1, which were overlooked or insufficiently handled by prior work. It decomposes MVM and Conv into composable units called *MetaKernels*, each encapsulating computation and data, requiring only one rotation. This design improves SIMD parallelism within ciphertexts (horizontal batching) and computational parallelism across ciphertexts (vertical batching). MetaKernel results are combined to produce outputs with consistent layouts, avoiding the costly layout conversions required by Fhelipe [Krastev et al. 2024].

Contributions. MKR significantly advances the practical deployment of large DNN models for encrypted inference. This paper makes the following major contributions:

- A composition-based compiler approach, MKR, that balances rotation overhead and slot utilization, significantly improving the efficiency of homomorphic MVM and Conv execution.
- The first unified framework for optimizing MVM and Conv using MetaKernels under CKKS.
- The first rotation-aware cost model for data packing in MVM and Conv.
- A comprehensive evaluation of MKR on real-world DNN models, including cases where ciphertexts exceed capacity, demonstrating performance gains over Fhelipe [Krastev et al. 2024], a state-of-the-art FHE compiler. MKR achieves 10.08×–185.60× speedups for MVM and Conv kernels and 1.75×–11.84× improvements in end-to-end inference.

Although our presentation focuses on CKKS, MKR generalizes to all RLWE-based FHE schemes, including RNS-CKKS [Cheon et al. 2019], BGV [Brakerski et al. 2011], and BFV [Fan and Vercauteren 2012], as it relies on a common set of three key FHE operations in RLWE-based schemes.

2 Background

We provide a brief overview of FHE and the CKKS scheme [Cheon et al. 2017], focusing on a set of three key FHE operations relevant to this work. A conceptual understanding of these operations is sufficient to follow our approach, without requiring deep knowledge of FHE intricacies.

2.1 Fully Homomorphic Encryption

FHE [Gentry 2009] enables quantum-safe computation on encrypted data without decryption, supporting PPML where the computing party remains blind to the data but can still generate encrypted results. A key PPML use case is secure task offloading to untrusted cloud servers.

Modern FHE schemes fall into two categories: Ring Learning With Errors (RLWE)-based (e.g., BGV [Brakerski et al. 2011], BFV [Fan and Vercauteren 2012], and CKKS [Cheon et al. 2017] (its RNS-CKKS variant [Cheon et al. 2019]) for arithmetic circuits and Learning With Errors (LWE)-based (e.g., GSW [Gentry et al. 2013], FHEW [Ducas and Micciancio 2015], and TFHE [Chillotti et al. 2018]) for boolean circuits, each targeting specific cryptographic challenges and computations.

RLWE-based schemes support SIMD parallelism, packing multiple elements into one ciphertext for efficiency. Homomorphic addition and multiplication operate element-wise on encrypted vectors, while rotation shifts encrypted elements left or right, making data manipulation costly.

2.2 The Cheon-Kim-Kim-Song Scheme

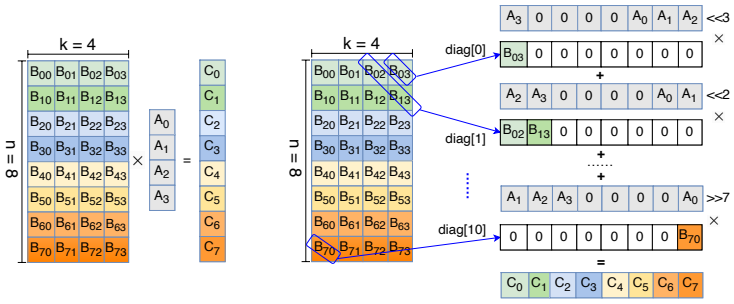
CKKS [Cheon et al. 2017], derived from BGV [Brakerski et al. 2011], supports approximate arithmetic on encrypted data. A cleartext vector of size $N/2$ is encoded and encrypted into a ciphertext over the ring $\mathbb{Z}_Q[X]/(X^N + 1)$, where N is the ring degree and Q is the ciphertext modulus. Each ciphertext holds $\mathcal{S} = N/2$ complex numbers. CKKS reduces the modulus size after multiplication through rescaling, similar to rounding in floating-point arithmetic, to control noise and maintain precision. Each multiplication consumes one modulus level, with the total levels L defined by the initial modulus $Q = q_0 \cdot \prod_{i=1}^{L} q_i$. The modulus at level ℓ is $Q^\ell = q_0 \cdot \prod_{i=1}^{\ell} q_i$. Once reduced to q_0, no further multiplication is possible without bootstrapping [Cheon et al. 2024; Krastev et al. 2024; Liu et al. 2025]. Q depends on the input scale Δ (for fixed-point arithmetic), output scale q_0 (for precision), and multiplicative depth (initial level L). Since operations at higher levels are more expensive, reducing multiplicative depth is important.

RNS-CKKS [Cheon et al. 2019] improves CKKS by using the Residue Number System (RNS) to decompose a polynomial at level ℓ with modulus Q^ℓ into $\ell + 1$ smaller polynomials with co-prime moduli $q_0, \ldots, q_\ell$, each representable by a 64-bit integer, improving arithmetic efficiency.

Due to its unique support for fixed-point arithmetic, CKKS—particularly its RNS-optimized variant—has been integrated into a number of FHE compilers [Boemer et al. 2019; Dathathri et al. 2020; Krastev et al. 2024; Li et al. 2025; Viand et al. 2022] to enable PPML [Lee et al. 2022a].

CKKS supports a limited set of operations: HAddCP (ciphertext-plaintext addition), HAddCC (ciphertext-ciphertext addition), HMulCP (ciphertext-plaintext multiplication), HMulCC (ciphertext-ciphertext multiplication), and HRot (ciphertext rotation). For MVM and Conv with plaintext weights, we focus on HAddCC, HMulCP, and HRot, following prior work [Krastev et al. 2024; Lee et al. 2022b]. Their time complexities under RNS-CKKS are $O(NL)$ for HAddCC and HMulCP, and $O(N \log NL^2)$ for HRot [Dathathri et al. 2019]. As HRot is one to two orders of magnitude costlier across different modulus levels [Cheon et al. 2024; Liu et al. 2025], reducing rotation overhead while maintaining high slot utilization is key to optimizing MVM and Conv.

To understand MKR, it suffices to know that HAddCC and HMulCP perform element-wise operations on encrypted vectors, while $\text{HRot}(c, s)$ cyclically shifts the vector in ciphertext c by $|s|$



(a) $C = B \times A$ (b) FHE–based Implementation

Fig. 1. FHE-MP-CNN's MVM solution ($\mathcal{S} = 8$): (a) $C = B \times A$ ($n = 8, k = 4$); (b) Homomorphic implementation by mapping 11 diagonals, `diag[0]` – `diag[10]`, of B to 11 distinct plaintexts and encrypting A as a ciphertext.

positions—left if $s \geqslant 0$, right if $s < 0$. In figures, $c \ll o$ and $c \gg o$ denote $\mathsf{HRot}(c, o)$ and $\mathsf{HRot}(c, -o)$ for $o \geqslant 0$. We write $\mathsf{HAddCC}_{k=0}^{n-1} c_k$ to represent homomorphic addition instead of $\sum_{k=0}^{n-1} c_k$.

3 Motivation

Our motivating kernel is MVM, $C = B \times A$, where B and A have shapes $[n, k]$ and $[k]$, respectively, producing C with shape $[n]$. Many DNN operations, such as fully connected layers in ResNet [He et al. 2015] and linear layers in LLMs [Meta 2024], map directly to MVM. In PPML, A is a ciphertext encrypting k elements, while B represents plaintext weights.

We begin with an MVM example where B and A have shapes $[8, 4]$ and $[4]$ ($n = 8, k = 4$) using a ciphertext with $\mathcal{S} = 8$ to introduce MetaKernels and highlight differences between MKR and state-of-the-art methods, FHE-MP-CNN [Lee et al. 2022b] and Fhelipe [Krastev et al. 2024]. We then extend to shapes $[8, 8]$ and $[8]$ ($n = 8, k = 8$) with $\mathcal{S} = 32$ to show how MetaKernels enable horizontal composition (batching) within ciphertexts (maximizing SIMD parallelism) and vertical composition (batching) across ciphertexts (exposing computational parallelism). These examples illustrate how MKR addresses Challenges **C1**–**C5** (Table 1) through a rotation-aware cost model that minimizes rotation overhead while ensuring high slot utilization. **C6** is discussed in Section 4.2.

In homomorphic MVM, we analyze the total rotations and slot utilization of B, defined as $(n \times k)/(\mathcal{S} \times n_{\text{ct}})$, where n_{ct} is the number of plaintexts used to pack B. Efficient packing of plaintext weights is crucial as it impacts the number of rotations needed to align with A.

Case 1: $n = 8$ and $k = 4$ ($\mathcal{S} = 8$). We first review the solutions from FHE-MP-CNN [Lee et al. 2022b] and Fhelipe [Krastev et al. 2024], then present our MetaKernel-based solution for this case.

FHE-MP-CNN. FHE-MP-CNN implements the fully connected layer of ResNet-20 as MVM using the SEAL library [SEAL 2020], as shown in Figure 1. It performs HMulCP between each diagonal vector of B and A, requiring A to be pre-rotated for alignment. This produces 11 intermediate ciphertexts, combined using HAddCC to generate C with the same layout as A. FHE-MP-CNN requires 10 rotations ($n + k - 2$) due to excessive zero padding. Increasing the slot size S does not reduce rotations since FHE-MP-CNN always packs one diagonal per plaintext, leading to poor slot utilization when S is large relative to n. The slot utilization rate of B in this example is 36.4%.

Fhelipe. Fhelipe [Krastev et al. 2024], a DSL compiler, automates data packing by fitting as many rows of B as possible into a plaintext to maximize slot utilization, replicating A to match the shape of B, as shown in Figure 2. It computes partial products using HMulCP, sums each row's products logarithmically using HAddCC (requiring row padding to a power of 2), and merges them to produce C. However, C ends up with a non-contiguous layout, often needing costly masking and rotations to restore it for subsequent layers. By fully replicating B, Fhelipe maximizes slot

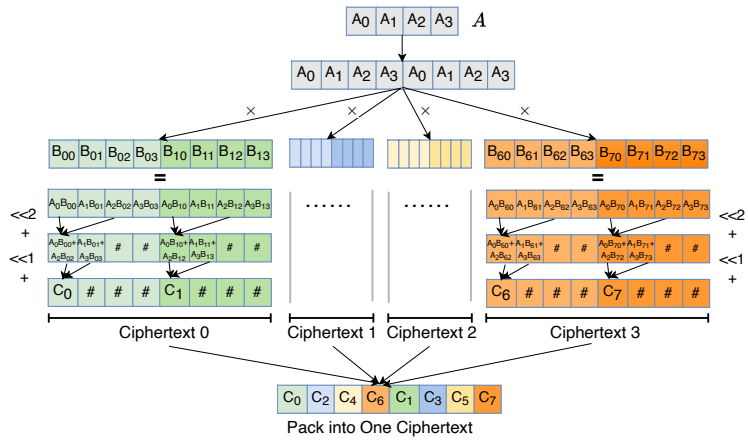


Fig. 2. Fhelipe's MVM solution ($\mathcal{S} = 8$) for the example in Figure 1(a).

utilization but overlooks rotation costs, degrading performance when $n \times k$ is large relative to $\mathcal{S}$. In this example, Fhelipe achieves 100% slot utilization for B at the cost of 12 rotations.

Our MKR Approach. We decompose MVM into *MetaKernels*, batching horizontally to improve slot utilization (SIMD) and vertically to expose computational parallelism. A rotation-aware cost model guides this to minimize rotation overhead while maintaining high slot utilization.

For MVM, we apply the Halevi–Shoup diagonal method [Shai and Victor 2014] to construct and optimize MetaKernels. For a vector v with encrypted elements, $\overline{\mathsf{HRot}}(v, s)$ denotes a circular shift of v by s positions. For instance, if $v = (v_0, v_1, v_2, v_3)$, then $\overline{\mathsf{HRot}}(v, 1) = (v_1, v_2, v_3, v_0)$ and $\overline{\mathsf{HRot}}(v, -1) = (v_3, v_0, v_1, v_2)$. $\overline{\mathsf{HRot}}(v, s)$ matches $\mathsf{HRot}(v, s)$ when v occupies a ciphertext of size $\mathcal{S} = |v|$, but becomes non-trivial when v is embedded in a larger ciphertext ($\mathcal{S} > |v|$). For example, embedding v in $c = (v_0, v_1, v_2, v_3, \#)$ yields $\mathsf{HRot}(c, 1) = (v_1, v_2, v_3, \#, v_0)$ and $\mathsf{HRot}(c, -1) = (\#, v_0, v_1, v_2, v_3)$. Thus, $\overline{\mathsf{HRot}}(v, 1)$ and $\overline{\mathsf{HRot}}(v, -1)$ cannot be realized using $\mathsf{HRot}(c, 1)$ and $\mathsf{HRot}(c, -1)$, respectively.

For a plaintext p, $p \ll o$ ($p \gg o$) denotes a circular left (right) shift of p by o for $o \geqslant 0$. For a matrix W (plaintext or ciphertext), $W \ll o$ ($W \gg o$) denote circularly shifting each row of W by o.

For a matrix W, $W[i]$ represents the i-th row, $W_{i,j}$ the element at row i and column j (with both starting from 0), and $W[i : j]$ the submatrix from columns i to j inclusive. If W is a (row) vector, $W[0] = W$, W_i denotes the i-th element, and $W[i : j]$ the sub-vector from i to j inclusive.

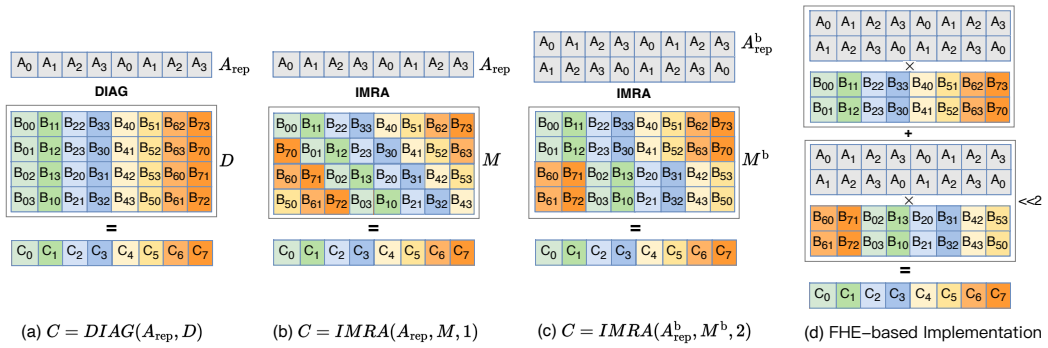
In Figure 3(a), we use the Halevi-Shoup diagonal method [Shai and Victor 2014] to compute $C = B \times A$ homomorphically by reformulating it as two sub-MVMs: $\begin{bmatrix} C_0 \\ C_1 \end{bmatrix} = \begin{bmatrix} B_0 \times A \\ B_1 \times A \end{bmatrix}$. Let $D = [D_0, D_1]$ (size 4×8), where D_i is the diagonal matrix of B_i (Section 4.1). Replicating A as $A_{\text{rep}} = [A, A]$ yields:

$$C = \mathsf{DIAG}(A_{\text{rep}}, D) = \mathsf{HAddCC}_{i=0}^{3}\mathsf{HMulCP}(\overline{\mathsf{HRot}}(A_{\text{rep}}, i), D[i]) \tag{1}$$

By packing B more efficiently as $M = [D[0] \gg 0; D[1] \gg 1; D[2] \gg 2; D[3] \gg 3]$ (where $[\cdot\,;\cdot]$ denotes vertical stacking of matrices[1]), we shift the necessary rotations from A (runtime) to B (compile-time), thereby reducing runtime overhead as illustrated in (Figure 3(b)):

$$C = \mathsf{IMRA}(A_{\text{rep}}, M, 1) = \mathsf{HAddCC}_{i=0}^{3}\overline{\mathsf{HRot}}(\mathsf{HMulCP}(A_{\text{rep}}, M[i]), i) \tag{2}$$

[1]In MATLAB, this corresponds to vertical concatenation using ";".



(a) $C = DIAG(A_{\text{rep}}, D)$ (b) $C = IMRA(A_{\text{rep}}, M, 1)$ (c) $C = IMRA(A^{\text{b}}_{\text{rep}}, M^{\text{b}}, 2)$ (d) FHE–based Implementation

Fig. 3. MKR's MVM solution ($\mathcal{S} = 8$) for the example in Figure 1(a): (a) DIAG (Equation (1)); (b) Unblocked IMRA ($bs = 1$); (c) Blocked IMRA ($bs = 2$); (d) FHE program generated from (c) with $\overline{\text{HRot}}$ replaced by HRot.

By vertically blocking M with block size $bs = 2$, we derive the following *Iterative Multiply–Rotate-Accumulate (IMRA)* pattern, which forms the basis for optimizing MVM and Conv (Figure 3(c)):

$$\begin{aligned} C &= \text{IMRA}(A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{bs \times col}, M^{\text{b}} \in \mathbb{R}^{row \times col}, Shift = bs) \\ &= \text{HAddCC}^{row/bs-1}_{g=0}\ \text{MetaKernel}(A^{\text{b}}_{\text{rep}}, M^{\text{b}}[(g \times bs : (g+1) \times bs - 1], Shift) \\ &= \text{HAddCC}^{row/bs-1}_{g=0}\ \overline{\text{HRot}}\Big(\text{HAddCC}^{bs-1}_{b=0}\text{HMulCP}(A^{\text{b}}_{\text{rep}}[b], M^{\text{b}}[g \times bs + b]), Shift\Big) \end{aligned} \tag{3}$$

For this example, $row = 4$ and $col = 8$. M^{b} is derived by circularly shifting M's odd rows by 1, and A_{rep} is expanded into a 2×8 matrix $A^{\text{b}}_{\text{rep}}$, with $A^{\text{b}}_{\text{rep}}[i] = \overline{\text{HRot}}(A_{\text{rep}}, i)$ to align with M^{b}. Intra-block shifts enable lockstep rotation for efficient execution. The unblocked IMRA pattern is a special case with $bs = 1$. Each MetaKernel performs bs HMulCP's, $bs - 1$ HAddCC's, and one rotation on $A^{\text{b}}_{\text{rep}}$ and block $M^{\text{b}}_i = M^{\text{b}}[i \times bs : i \times bs + bs - 1]$. There are four MetaKernels if $bs = 1$ and two if $bs = 2$.

The IMRA pattern explores different layouts for B by composing MetaKernels—horizontally to improve slot utilization (SIMD parallelism) and vertically to enhance computational parallelism. Unlike prior work, MKR uses rotation-aware data packing in MetaKernel composition, enabling a fast exhaustive search to minimize rotations first and maximize slot utilization in case of ties.

Figure 3(d) shows MKR's solution, derived from Figure 3(c) (or indirectly from Figure 3(b) via Figure 3(c)). Since $\mathcal{S} = 8$, each row of $A^{\text{b}}_{\text{rep}}$ and M^{b} fits within a ciphertext and plaintext, respectively, allowing $\overline{\text{HRot}}$ to be replaced by HRot. The two MetaKernels, each performing two HMulCP's, one HAddCC, and one rotation, are composed vertically. MKR achieves 100% slot utilization for B with only 3 rotations—the minimum possible—far fewer than FHE-MP-CNN and Fhelipe.

If Figure 3(b) (unblocked) is used as the solution, 4 rotations are required despite achieving 100% slot utilization for B. In general, transitioning from an unblocked to a blocked layout with block size bs reduces intra-block rotations from bs to 1 but incurs additional rotations for expanding A_{rep}—a trade-off that can be systematically evaluated using our rotation-aware cost model. Since Figure 3(c) reduces the total rotation count by one, MKR favors it over the unblocked pattern.

Case 2: $n = 8$ and $k = 8$ ($\mathcal{S} = 32$). In this larger MVM example, FHE-MP-CNN requires 14 rotations with 13.3% slot utilization for B, while Fhelipe requires 9 rotations with 100% slot utilization. FHE-MP-CNN preserves C's layout, but Fhelipe scatters C's 8 elements across a vector of size 32, | C0 | C4 | # | ... | # | C1 | C5 | # | ... | # | C2 | C6 | # | ... | # | C3 | C7 | # | ... | # |, requiring rotations to restore its original layout.

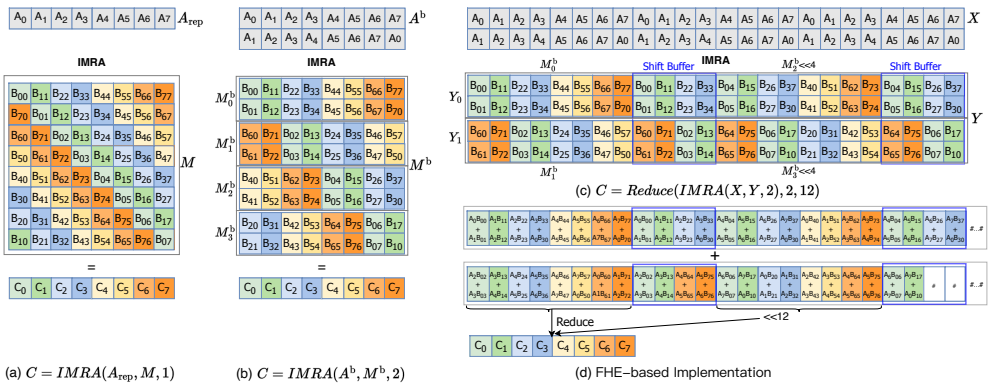


Fig. 4. MKR's MVM solution ($\mathcal{S} = 32$) for $C = B \times A$ ($n = k = 8$): (a) IMRA($A_{\text{rep}} \in \mathbb{R}^{1\times8}, M \in \mathbb{R}^{8\times8}, 1$); (b) IMRA($A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{2\times8}, M^{\text{b}} \in \mathbb{R}^{8\times8}, 2$); (c) IMRA($X \in \mathbb{R}^{2\times24}, Y \in \mathbb{R}^{4\times24}, 2$); (d) FHE program generated from (c).

To improve slot utilization, MKR applies blocking to a given IMRA pattern (unblocked or pre-blocked), enabling vertical (as in Figure 3(d)) and horizontal composition of MetaKernels (as shown here). Horizontal composition packs and concatenates data blocks into a larger block, forming a single MetaKernel. For MVM, this involves embedding blocks of B and replicating A for alignment. To synthesize an FHE program from the optimized pattern, all logical $\overline{\text{HRot}}$ rotations in horizontally composed MetaKernels must be replaced by HRot —a challenge our approach addresses.

Figure 4 illustrates MKR's optimal solution. Figures 4(a) and (b) depict C=IMRA($A_{\text{rep}} \in \mathbb{R}^{1\times8}, M \in \mathbb{R}^{8\times8}, Shift = bs = 1$) and C = IMRA($A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{2\times8}, M^{\text{b}} \in \mathbb{R}^{8\times8}, Shift = bs = 2$), analogous to Figures 3(b) and (c). Figure 4(c) shows the optimized IMRA pattern layout, C=$Reduce$(IMRA($X \in \mathbb{R}^{2\times24}, Y \in \mathbb{R}^{4\times24}, Shift = bs = 2$), 2, 12) (with two MetaKernels), executed homomorphically in Figure 4(d) with $\overline{\text{HRot}}$ replaced by HRot. For a ciphertext Z, the reduction operation is defined as:

$$Reduce(Z, r, Shift) = \begin{cases} \text{HAddCC}_{i=0}^{r-1}\ \text{HRot}(Z, i \times Shift) & r > 1 \\ Z & r = 1 \end{cases} \tag{4}$$

For this example, $r = 2$ denotes the number of horizontally batched MetaKernels, requiring result reduction. The first eight elements of C, $[C_0, C_1, \ldots, C_7]$, match the expected layout.

Here, we transform Figure 4(b) (with four blocks M^{b}_0–M^{b}_3) into Figure 4(c) (with two blocks Y_0 and Y_1). X is derived by replicating $A^{\text{b}}_{\text{rep}}$ three times. By Equation (3), the $\overline{\text{HRot}}$ offsets for M^{b}_0, M^{b}_1, M^{b}_2, and M^{b}_3 are 0, 2, 4, and 6, respectively, with the blocks cyclically distributed across Y_0 and Y_1.

For M^{b}_2, instead of placing it directly in Y_0, we embed $M^{\text{b}}_2 \ll 4$, aligning it with the second A_4 position in the first row of X. This avoids a $\overline{\text{HRot}}$ rotation since $4 - 4 = 0$, like M^{b}_0 at the start of Y_0. Similarly, $M^{\text{b}}_3 \ll 4$ is embedded in Y_1, requiring a $\overline{\text{HRot}}$ rotation of $6 - 4 = 2$, the same as M^{b}_1 in Y_1.

Directly implementing $\overline{\text{HRot}}$ for M^{b}_1 and $M^{\text{b}}_3 \ll 4$ (offset 2 in Y_1) would be costly. However, cyclic embedding creates an unused 4-element gap, repurposed as a *shift buffer* for redundant computation of the first four elements in each block (only the first two are needed). As shown in Figures 4(c) and (d), HRot can now efficiently handle the two logical $\overline{\text{HRot}}$ rotations using this shift buffer.

Our solution requires 5 rotations (lowest possible) and achieves 50.0% slot utilization for B.

4 The MetaKernel-Based Design

We now present MKR, a new approach for synthesizing high-performance homomorphic MVM and Conv kernels using the MetaKernel-based IMRA pattern, addressing Challenges **C1**–**C6** (Table 1).

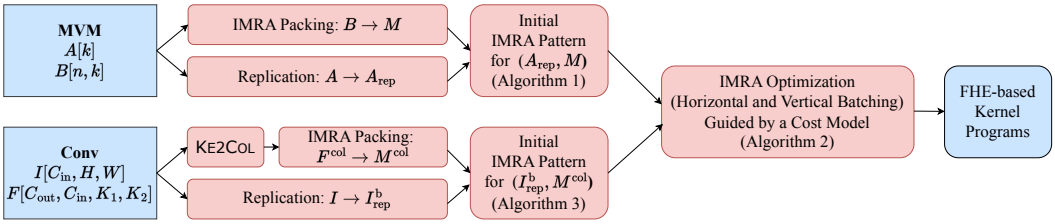


Fig. 5. MKR's approach for optimizing homomorphic MVM and Conv using a MetaKernel-based framework.

Each MetaKernel encapsulates computation and data, requiring one rotation. MetaKernels are composed vertically and horizontally via blocking and repacking, forming rotation-aware IMRA layouts. MKR optimally balances rotation overhead and slot utilization, achieving SIMD parallelism within ciphertexts (horizontal batching) and computational parallelism across them (vertical batching).

Figure 5 shows how MKR maps MVM and Conv kernels into FHE programs. Starting from initial IMRA layouts (Algorithm 1 for MVM and Algorithm 3 for Conv), MKR optimizes them using the same horizontal and vertical batching process (Algorithm 2). We introduce *ke2col*, a novel approach for computing Conv homomorphically, enabling a unified framework for both MVM and Conv.

Therefore, we describe our approach as follows. Section 4.1 focuses on MVM, emphasizing IMRA pattern optimization (Algorithm 2), as the basics of Algorithm 1 are already introduced when motivating our approach in Section 3. Section 4.2 introduces the initial IMRA layout for Conv (Algorithm 3), unifying MVM and Conv optimization through reuse of Algorithm 2. Section 4.3 compares MKR with prior work in terms of rotation overhead and slot utilization.

To generate FHE programs for MVM and Conv, we start with an IMRA pattern where all $\overline{\text{HRot}}$ rotations are logical, as motivated in Section 3 for MVM. We will refine it into an optimized pattern where $\overline{\text{HRot}}$ is replaced by HRot for homomorphic execution with slot size $\mathcal{S}$.

An FHE program derived from an IMRA pattern layout (or configuration) for an MVM or Conv kernel $\mathcal{K}$ is *correct* if it produces the encrypted form of the same output as the unencrypted kernel. Two IMRA pattern configurations of $\mathcal{K}$ are *equivalent* if they yield the same encrypted output.

Let W be a matrix of size $r \times s$ (ciphertext). HReplicate(W, rep) creates a matrix $[W, W, \ldots, W]$ of size $r \times (s \times rep)$ by horizontally replicating W rep times. VReplicate$(W, rep, alignment)$ creates a matrix $[W_0; W_1; \ldots; W_{rep-1}]$ of size $(r \times rep) \times s$ by vertically replicating W rep times, with each row of W_i circularly left-shifted by $alignment[i]$: $W_i = W \ll alignment[i]$.

4.1 Matrix-Vector Multiplication (MVM)

We present our algorithms for generating a high-performance homomorphic kernel for $C = B \times A$, where B and A have shapes $[n, k]$ and $[k]$, respectively, with slot size $\mathcal{S}$. B must be *shape-divisible* ($n \,\%\, k = 0 \vee k \,\%\, n = 0$); otherwise, both are zero-padded. We assume each row and column fits into $\mathcal{S}$, which is reasonable since $\mathcal{S} \geqslant$ 32K is typical [Cheon et al. 2017; Samardzic et al. 2021, 2022].

In Section 4.1.1, we formalize the unblocked IMRA pattern (Section 3). In Section 4.1.2, we optimize it through blocking, enabling horizontal and vertical MetaKernel composition.

4.1.1 Unblocked IMRA Pattern. Following the motivation in Section 3, we apply the Halevi-Shoup diagonal method [Shai and Victor 2014] to initially create the unblocked IMRA pattern. For a square matrix $U = (U_{i,j})_{0 \leqslant i,j < m}$, we extract its *generalized diagonals* to form $\mathscr{D}(U)$:

$$\mathscr{D}(U) = [\text{diag}_0; \text{diag}_1; \ldots; \text{diag}_{m-1}], \text{ where } \text{diag}_i = (U_{0,i}, U_{1,(i+1)\,\%\,m}, \ldots, U_{m-1,(i+m-1)\,\%\,m}) \tag{5}$$

Algorithm 1: Code generation for MVM with slot size $\mathcal{S}$ using the unblocked IMRA pattern.

Input: $B \in \mathbb{R}^{n\times k}$ (plaintext) and $A \in \mathbb{R}^{k}$ (ciphertext).
Output: FHE program for computing C(ciphertext) $= B \times A$ homomorphically
1 $M \in \mathbb{R}^{n_d \times k_d} \leftarrow$ IMRAPacking(B), where $n_d = \min(n,k)$ and $k_d = \max(n,k)$;
2 $A_{\text{rep}} \leftarrow$ HReplicate(A, rep $= \frac{k_d}{k}$);
3 *OptPat* $\leftarrow$ Optimize_IMRA("Z_{IMRA} = IMRA($A_{\text{rep}} \in \mathbb{R}^{k_d}, M \in \mathbb{R}^{n_d \times k_d}$, *Shift* = 1)");
4 Generate the following FHE-based kernel code:

5
OptPat from line 16 of Algorithm 2 (where Z_{IMRA} stores the computed result) ; // 5a
$C_{IMRA} \leftarrow$ *Reduce*($Z_{\text{IMRA}}, \frac{k}{n}, n$) ; // 5b
$C \leftarrow$ HMulCP($C_{\text{IMRA}}, mask[1]_n$) ; // 5c

Let $n_d = \min(n,k)$ and $k_d = \max(n,k)$. We partition B into k_d/n_d square matrices, $B_0, B_1, \ldots, B_{k_d/n_d-1}$, along dimension n for tall matrices ($n \geqslant k$) and k for short ones ($n < k$). Diagonalizing these matrices gives $D = [\mathscr{D}(B_0), \mathscr{D}(B_1), \ldots, \mathscr{D}(B_{k_d/n_d-1})]$. To align A with D, we use HReplicate($A, k_d/k$) to replicate A k_d/k times, forming A_{rep} (a vector of length k_d). Thus,

$$C = B \times A = \text{DIAG}(A_{\text{rep}}, D) = \textit{Reduce}(\text{HAddCC}_{i=0}^{n_d-1}\, \text{HMulCP}(\overline{\text{HRot}}(A_{\text{rep}}, i), D[i]), \frac{k}{n}, n) \quad (6)$$

where *Reduce* is defined in Equation (4). The i-th sub-vector $A^i_{\text{rep}} = A_{\text{rep}}[i \times n_d : (i+1) \times n_d - 1]$ is used in the i-th sub-MVM calculation $B_i \times A^i_{\text{rep}}$. For short matrices ($n < k$), the reduction step combines the partial products from the sub-MVM calculations.

Let M=IMRAPacking(B)=$\mathscr{M}(\mathscr{D}(B))$=$\mathscr{M}(D)$ be a rotation-efficient data packing for B, where

$$\mathscr{M}(D) = [D[0] \gg 0; D[1] \gg 1; \ldots; D[n_d - 1] \gg (n_d - 1)] \quad (7)$$

As shown below, this enables transferring rotations from A (runtime) to B (compile-time), yielding:

$$C = B \times A = \textit{Reduce}(\text{HAddCC}_{i=0}^{n_d-1}\, \overline{\text{HRot}}(\text{HMulCP}(A_{\text{rep}}, M[i], i)), \frac{k}{n}, n) \quad (8)$$

We can now express homomorphic MVM using the following *unblocked IMRA pattern*:

$$\begin{aligned} Z_{\text{IMRA}} &= \text{IMRA}(A_{\text{rep}} \in \mathbb{R}^{k_d}, M \in \mathbb{R}^{n_d \times k_d}, \textit{Shift} = 1) \\ &= \text{HAddCC}_{i=0}^{n_d-1}\text{MetaKernel}(A_{\text{rep}}, M[i], i \times \textit{Shift}) \\ &= \text{HAddCC}_{i=0}^{n_d-1}\overline{\text{HRot}}\left(\text{HMulCP}(A_{\text{rep}}, M[i]), i \times \textit{Shift}\right) \\ C_{\text{IMRA}} &= \textit{Reduce}(Z_{\text{IMRA}}, \frac{k}{n}, n) \end{aligned} \quad (9)$$

This initial unblocked pattern consists of n_d MetaKernels, each performing one HMulCP and requiring one rotation. This leads to Algorithm 1, which constructs an FHE-based MVM kernel and applies Optimize_IMRA from Algorithm 2 for further optimization (lines 1–3). In the final program (lines 4–5), the result C is extracted from the first n elements of C_{IMRA} (line 5c).

We revisit the two motivating MVM examples from Section 3. For $B([8,4])$ and $A([4])$, we illustrated DIAG(A_{rep}, D) in Equation (6) and the unblocked IMRA pattern in Figure 3(a) and Figure 3(b), respectively. For $B([8,8])$ and $A([8])$, the unblocked IMRA pattern is shown in Figure 4(a).

Lemma 4.1. *Let OptPat represent the unblocked IMRA pattern (Equation (9)) itself in line 3 of Algorithm 1 (bypassing Algorithm 2). Then the FHE kernel generated by Algorithm 1 for MVM is correct, even if* $\overline{\text{HRot}}$ *in the unblocked IMRA pattern is replaced by* HRot, *provided* $\mathcal{S} = k_d$.

Proof. For $C = B \times A$, with B plaintext and A ciphertext, the Halevi-Shoup diagonal method [Shai and Victor 2014], as specified in Equation (6), correctly computes MVM. The transformations from Equation (6) to Equation (9) preserve correctness due to rotational invariance on B and A, which is guaranteed by construction. When $\mathcal{S} = k_d$, $\overline{\text{HRot}}$ and HRot are functionally identical. □

4.1.2 Optimized IMRA Pattern. For MVM, we begin with the unblocked IMRA pattern (line 3 of Algorithm 1) and invoke Optimize_IMRA from Algorithm 2 to select an optimal blocking strategy. For Conv, we start with a pre-blocked IMRA pattern due to the nature of this kernel, as will be detailed in Section 4.2. To handle both cases, Optimize_IMRA is designed to apply additional blocking, even to patterns that have already been blocked.

Below, we first derive the blocked pattern from the unblocked version, treating the latter as a special case. Next, we apply Optimize_IMRA to refine it, enabling horizontal and vertical MetaKernel composition. Finally, we identify optimal blocking parameters using a rotation-aware cost model.

(a) Blocked IMRA Pattern. Given the unblocked pattern $\text{IMRA}(A_{\text{rep}} \in \mathbb{R}^{k_d}, M \in \mathbb{R}^{n_d \times k_d}, \mathit{Shift} = 1)$ (Equation (9)), we block it by dividing M into n_d/bs blocks along dimension n_d, each with bs rows. We define $M^{\text{b}}[i] = M[i] \ll (i \,\%\, bs)$, shifting its i-th row within each block by $i \,\%\, bs$. We expand A_{rep} into $A^{\text{b}}_{\text{rep}}$ (size $bs \times k_d$) using $\text{VReplicate}(A_{\text{rep}}, bs, [0, 1, ..., bs-1])$, setting $A^{\text{b}}_{\text{rep}} = [A_{\text{rep}} \ll 0; A_{\text{rep}} \ll 1; \ldots; A_{\text{rep}} \ll (bs-1)]$ to align with M^{b}. This yields the following *blocked IMRA pattern*:

$$\begin{aligned} C_{\text{IMRA}} &= \text{IMRA}(A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{bs \times k_d}, M^{\text{b}} \in \mathbb{R}^{n_d \times k_d}, \mathit{Shift} = bs) \\ &= \text{HAddCC}^{n_d/bs-1}_{g=0}\text{MetaKernel}(A^{\text{b}}_{\text{rep}}, M^{\text{b}}[g \times bs : (g+1) \times bs - 1], g \times \mathit{Shift}) \\ &= \text{HAddCC}^{n_d/bs-1}_{g=0}\overline{\text{HRot}}\left(\text{HAddCC}^{bs-1}_{b=0}\text{HMulCP}(A^{\text{b}}_{\text{rep}}[b], M^{\text{b}}[g \times bs + b]), g \times \mathit{Shift}\right) \end{aligned} \tag{10}$$

where the i-th MetaKernel operates on the i-th block of M^{b}, $M^b_i = M^{\text{b}}[i \times bs : (i+1) \times bs - 1]$. For convenience, we sometimes refer to all such M^b_i blocks as M^b*-blocks*.

The unblocked pattern specified in Equation (9) is a special case where $bs = 1$.

Revisiting the two MVM examples from Section 3, we illustrated the blocked pattern for $B([8,4])$ and $A([4])$ in Figure 3(c) and for $B([8,8])$ and $A([8])$ in Figure 4(b), both with $bs = 2$.

(b) Horizontal and Vertical Batching. In Algorithm 2, we optimize a blocked IMRA pattern $\widehat{C_{\text{IMRA}}} = \text{IMRA}(\widehat{A^{\text{b}}_{\text{rep}}} \in \mathbb{R}^{bs \times k_d}, \widehat{M^{\text{b}}} \in \mathbb{R}^{n_d \times k_d}, \mathit{Shift})$, by replacing $\overline{\text{HRot}}$ with HRot to reduce the number of rotations, enabling direct generation of an FHE-based kernel (line 16). The optimization selects a parameter pair (P_b, P_s) (line 1) to further partition $\widehat{M^{\text{b}}}$ into P_b superblocks, each consisting of bs^{opt}/bs blocks in the "Superblocking" step (lines 2–7). Next, we batch P_s superblocks horizontally, fusing their MetaKernels into larger ones and yielding $gs = P_b/P_s$ vertically stacked MetaKernels in the "Composition" step (lines 8–16). We now describe how to generate an IMRA pattern layout for a given (P_b, P_s) pair, deferring the discussion of how to select optimal blocking parameters.

- *Superblocking (lines 2–7):* Conceptually, this step applies blocking to an IMRA pattern that has already been blocked. We first vertically stack all P_b superblocks—temporarily ignoring P_s—to form a superblocked IMRA pattern (line 7) with an updated superblock-level rotation offset $\mathit{Shift}^{\text{opt}}$ (line 6). Each M^{b}-superblock consists of bs^{opt} rows of $\widehat{M^{\text{b}}}$, or equivalently, $\frac{bs^{\text{opt}}}{bs}$ $\widehat{M^{\text{b}}}$-blocks, where bs^{opt} is defined in line 2. To ensure synchronized rotations, each superblock applies appropriate shifts to its constituent $\widehat{M^{\text{b}}}$-blocks (lines 4–5). We also expand $\widehat{A^{\text{b}}_{\text{rep}}}$ to $A^{\text{b}}_{\text{rep}}$ of size $bs^{\text{opt}} \times k_d$ to align with the superblock structure (line 3).
- *Composition (lines 8-16):* After creating the super-blocked IMRA pattern (line 7), we horizontally batch P_s superblocks, merging their MetaKernels into a larger one. We then vertically assemble $gs = P_b/P_s$ of these enlarged MetaKernels to finalize the optimized pattern (line 16). Each i-th

Algorithm 2: OPTIMIZE_IMRA: Optimizing a blocked IMRA pattern.

Input: $\widehat{C}_{\text{IMRA}} = \text{IMRA}(\widehat{A}^{\text{b}}_{\text{rep}} \in \mathbb{R}^{bs \times k_d}, \widehat{M}^{\text{b}} \in \mathbb{R}^{n_d \times k_d}, \mathit{Shift})$

Output: FHE program for computing $C(\text{ciphertext}) = B \times A$ homomorphically

1 $(P_b, P_s) \leftarrow$ optimal parameters from minimizing the cost as specified in Equation (11) ;

2 $bs_{\text{opt}} \leftarrow \frac{n_d}{P_b}$;

3 $A^{\text{b}}_{\text{rep}} \leftarrow \text{VREPLICATE}(\widehat{A}^{\text{b}}_{\text{rep}}, \frac{bs^{\text{opt}}}{bs}, [0 \times \mathit{Shift}, 1 \times \mathit{Shift}, ..., (\frac{bs^{\text{opt}}}{bs} - 1) \times \mathit{Shift}])$;

4 **for** $0 \leqslant i < P_b$ **do**

5 $\quad M^{\text{b}}_i = [\widehat{M^{\text{b}}}_{i \times f} \ll 0 \times \mathit{Shift}; \widehat{M^{\text{b}}}_{i \times f+1} \ll 1 \times \mathit{Shift}; ...; \widehat{M^{\text{b}}}_{i \times f+f-1} \ll ((f-1) \times \mathit{Shift})]$, where $f = \frac{bs^{\text{opt}}}{bs}$

6 $\mathit{Shift}_{\text{opt}} \leftarrow \frac{bs^{\text{opt}}}{bs} \times \mathit{Shift}$;

7 Create a superblocked IMRA pattern: $C^{\text{b}}_{\text{IMRA}} = \text{IMRA}(A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{bs^{\text{opt}} \times k_d}, M^{\text{b}} \in \mathbb{R}^{n_d \times k_d}, \mathit{Shift}^{\text{opt}})$;

8 $gs \leftarrow \frac{P_b}{P_s}$;

9 $X \in \mathbb{R}^{\frac{n_d}{P_b} \times (P_s+1)} \leftarrow \text{HREPLICATE}(A^{\text{b}}_{\text{rep}}, P_s + 1)$;

10 $s_f \leftarrow gs \times \mathit{Shift}^{\text{opt}}$;

11 **for** $0 \leqslant p < P_b$ **do**

12 $\quad \underline{M^{\text{b}}}_p \leftarrow M^{\text{b}}_p \ll ((p \div gs) \times s_f)$

13 **for** $0 \leqslant i < gs$ **do**

14 $\quad Y_i \leftarrow \left[\underline{M^{\text{b}}_i} M^{\text{b}}_i[0 : s_f - 1]\ \underline{M^{\text{b}}}_{i+gs} \underline{M^{\text{b}}}_{i+gs}[0 : s_f - 1] ... \underline{M^{\text{b}}}_{i+gs(P_s-1)} \underline{M^{\text{b}}}_{i+gs(P_s-1)}[0 : s_f - 1]]\right]$;

15 Generate the following FHE-based kernel code:

$$Z_{\text{IMRA}} = \text{IMRA}(X \in \mathbb{R}^{bs^{\text{opt}} \times (k_d \times (P_s+1))}, Y \in \mathbb{R}^{\frac{n_d}{P_s} \times (k_d \times (P_s+1))}, \mathit{Shift}^{\text{opt}} = \frac{bs^{\text{opt}}}{bs} \times \mathit{Shift})\ ; \quad \text{// 16a}$$

$$= \text{HAddCC}_{g=0}^{gs-1} \text{MetaKernel}(X, Y[g \times bs^{\text{opt}} : (g+1) \times bs^{\text{opt}} - 1], g \times \mathit{Shift}^{\text{opt}})\ ; \quad \text{// 16b}$$

$$= \text{HAddCC}_{g=0}^{gs-1} \text{HRot}\left(\text{HAddCC}_{b=0}^{bs^{\text{opt}}-1} \text{HMulCP}(X[b], Y[g \times bs^{\text{opt}} + b]), g \times \mathit{Shift}^{\text{opt}}\right)\ ; \quad \text{// 16c}$$

$$C_{\text{IMRA}} = \mathit{Reduce}(Z_{\text{IMRA}}, P_s, k_d + s_f)\ ; \quad \text{// 16d}$$

16

MetaKernel processes $X \in \mathbb{R}^{bs^{\text{opt}} \times (P_s+1)}$ and the corresponding block $Y_i \in \mathbb{R}^{\frac{n_d}{P_s} \times (P_s+1)}$, where $Y_i = Y[i \times bs^{\text{opt}} : (i+1) \times bs^{\text{opt}} - 1]$ according to our convention. X results from replicating $A^{\text{b}}_{\text{rep}}$ horizontally $P_s + 1$ times, while Y_i is obtained by cyclic embedding from M^{b} (lines 10-14).

By cyclically distributing P_b M^{b}-blocks, $M^{\text{b}}_0, M^{\text{b}}_1, \ldots, M^{\text{b}}_{P_b-1}$, across $Y_0, Y_1, \ldots, Y_{gs-1}$, we embed sequences $\underline{M^{\text{b}}}_i, \underline{M^{\text{b}}}_{i+gs}, \ldots, \underline{M^{\text{b}}}_{i+gs \times (P_s-1)}$ into Y_i (line 14), where $0 \leq i < gs$. Here, $\underline{M^{\text{b}}}_{i+gs \times j}$ is defined as $M^{\text{b}}{}_{i+gs \times j}$ shifted by $j \times s_f$ (line 12), with s_f specified in line 10.

The $\overline{\text{HRot}}$ rotation offset for $M^{\text{b}}{}_{i+j \times gs}$ is $(j \times gs + i) \times \mathit{Shift}^{\text{opt}} = j \times s_f + i \times \mathit{Shift}^{\text{opt}}$ (from the superblocked pattern in line 7). Embedding $\underline{M^{\text{b}}}_{i+j \times gs}$ at $j \times (k_d + s_f)$ in Y_i aligns it with $X[j \times (k_d + s_f) : j \times (k_d + s_f) + k_d - 1] = A^{\text{b}}_{\text{rep}} \ll (j \times s_f)$, as enabled by HREPLICATE (line 9).

This layout reduces the rotation offset for each $M^{\text{b}}{}_{i+gs \times j}$ from $j \times s_f + i \times \mathit{Shift}^{\text{opt}}$ to $i \times \mathit{Shift}^{\text{opt}}$, ensuring a consistent $\overline{\text{HRot}}$ rotation offset $i \times \mathit{Shift}^{\text{opt}}$ across all P_s $\underline{M^{\text{b}}}$-blocks embedded in Y_i, reaching $(gs-1) \times \mathit{Shift}^{\text{opt}}$ when $i = gs - 1$. Each $\underline{M^{\text{b}}}$-block ends with an s_f-sized gap, repurposed as a *shift buffer* of size $s_f = gs \times \mathit{Shift}^{\text{opt}}$ to enable efficient $\overline{\text{HRot}}$ rotations via HRot through redundant computation. The buffer is thus large enough to handle all required rotation offsets.

As shown in line 14, the first s_f section of $\underline{M^{\text{b}}}_{i+j \times gs}$ is replicated at its end, aligning with a similar replication in X enabled by HREPLICATE (line 9). This produces the final optimized layout

(16a) with P_s MetaKernels (16b). Rotating Y_i by $i \times Shift^{\text{opt}}$ using HRot (instead of $\overline{\text{HRot}}$) ensures uniform rotation across all $\underline{M^{\text{b}}}$-blocks embedded in Y_i (16c), enabling accumulation with HAddCC (line 16c) and summarization with *Reduce* (line 16d) from Equation (4).

Let us illustrate how Optimize_IMRA from Algorithm 2 works using Figure 4 for MVM with $n = k = 8$ ($\mathcal{S} = 32$), yielding $n_d = k_d = 8$. Starting with the unblocked IMRA pattern in Figure 4(a), represented as $\widehat{C}_{\text{IMRA}} = \text{IMRA}(\widehat{A^{\text{b}}_{\text{rep}}} \in \mathbb{R}^{1\times 8}, \widehat{M^{\text{b}}} \in \mathbb{R}^{8\times 8}, Shift = bs = 1)$, MKR optimizes it in two steps. Using optimal parameters $(P_b, P_s) = (4, 2)$ (line 1), we obtain $bs^{\text{opt}} = 2$ and $Shift^{\text{opt}} = 2$.

In the superblocking step, MKR generates the superblocked IMRA pattern $C_{\text{IMRA}} = \text{IMRA}(A^{\text{b}}_{\text{rep}} \in \mathbb{R}^{2\times 8}, M^{\text{b}} \in \mathbb{R}^{8\times 8}, Shift^{\text{opt}} = bs^{\text{opt}} = 2)$, shown in Figure 4(b). In the composition step, with $gs = P_b/P_s = 2$, we derive $C_{\text{IMRA}} = Reduce(\text{IMRA}(X \in \mathbb{R}^{2\times 24}, Y \in \mathbb{R}^{4\times 24}, Shift^{\text{opt}} = 2), 2, 12)$ (line 16), illustrated in Figure 4(c). This depicts the cyclic embedding of M^{b}_0–M^{b}_3 across Y_0 and Y_1. The shift buffer size is $s_f = gs \times Shift^{\text{opt}} = 4$. $A^{\text{b}}_{\text{rep}}$ is replicated $P_s + 1 = 3$ times. M^{b}_2 is circularly left-shifted by $s_f = 4$ to align with X, with its first half replicated at the end. Similarly, M^{b}_3 is left-shifted by $s_f = 4$ and embedded in Y_1, with its first half redundantly replicated.

Lemma 4.2. *In Algorithm 2, the input and output IMRA patterns are equivalent.*

Proof. The superblocking phase (lines 2–7) preserves equivalence since the blocking process inherently maintains the same computation between the superblocked pattern (line 7) and the input (lines 2–6). In the composition phase (lines 8–16), cyclic embedding (lines 8–14) ensures that each MetaKernel in the superblocked pattern, once composed in the output pattern (line 16), produces results identical to the original, thereby preserving equivalence. □

(c) Optimal Blocking (Batching). Among the three primitives—HAddCC, HMulCP, and HRot—HRot is the most expensive, often one to two orders of magnitude costlier across modulus levels [Cheon et al. 2024; Liu et al. 2025]. We optimize MVM and Conv by minimizing HRot at the operator level, while rescaling and bootstrapping are typically applied globally at the model level in DNNs using our kernels [Cheon et al. 2024; Krastev et al. 2024; Liu et al. 2025].

A blocked IMRA pattern layout enables a rotation-aware cost model to identify optimal blocking parameters (P_b, P_s) (line 1 of Algorithm 2) via fast exhaustive search, minimizing total rotations and, in case of ties, maximizing slot utilization:

$$\textbf{Minimize} \underbrace{\underbrace{\gamma(rot(\widehat{A^{\text{b}}_{\text{rep}}}) + P_s + 1)}_{\text{One Row}} + \underbrace{\frac{n_d}{P_b} - 1}_{\text{Row Alignment}}}_{\text{Input Cost for Replicating } X \text{ from } \widehat{A^{\text{b}}_{\text{rep}}}} + \underbrace{\frac{P_b}{P_s} - 1}_{\text{Shift}} + \underbrace{P_s - 1}_{\text{Reduction}} \tag{11}$$

$$\textbf{subject to} \quad (1)\ n_d \,\%\, P_b = 0, (2)\ \frac{n_d}{P_b} \,\%\, bs = 0, (3)\ P_b \,\%\, P_s = 0, \ \textbf{and}\ (4)\ k_d \times (P_s + 1) \leqslant \mathcal{S}$$

The objective function includes "Shift" for $(\frac{P_b}{P_s} - 1)$ HRot rotations (line 16c) and "Reduce" for $P_s - 1$ rotations (line 16d). Preparing X involves vertically replicating $\widehat{A^{\text{b}}_{\text{rep}}}$ into $A^{\text{b}}_{\text{rep}}$ (line 3) and horizontally replicating $A^{\text{b}}_{\text{rep}}$ $P_s + 1$ times (line 9) to achieve dimensions $\frac{P_b}{P_s} \times (k_d \times (P_s + 1))$. Let $rot(\widehat{A^{\text{b}}_{\text{rep}}})$ denote the rotations needed to obtain one row of $\widehat{A^{\text{b}}_{\text{rep}}}$ from A. The rotation cost to derive X from A includes $\gamma(rot(\widehat{A^{\text{b}}_{\text{rep}}}) + P_s + 1)$ for the first row and $\frac{n_d}{P_b} - 1$ for aligning subsequent rows. Here, $\gamma(m)$ computes rotations for replicating A m times as the sum of bit positions of 1's in m's binary decomposition plus the number of 1's minus 1. For example, $\gamma(9) = 3(1001) + 2 - 1 = 4$.

Constraints ensure: (1) superblocks divide n_d, (2) each M^{b}-superblock holds complete $\widehat{M^{\mathrm{b}}}$-blocks, (3) MetaKernels process equal superblocks (line 16b), and (4) a row of Y fits in one ciphertext.

(d) Generating Optimal FHE-based MVM Kernels. For MVM, we start with the unblocked pattern from Algorithm 1 and then apply Algorithm 2 to generate a rotation-optimized FHE kernel.

Theorem 4.3. *The FHE-based kernel program generated by Algorithm 2 for MVM is correct.*

Proof. The correctness follows directly from Lemmas 4.1 and 4.2. □

4.2 Convolution (Conv)

We describe how MKR generates high-performance homomorphic Conv implementations (with an encrypted image and plaintext kernel weights) using our MetaKernel-based framework (Figure 5), addressing Challenges **C1**–**C6** (Table 1). While the classic Im2Col approach [Jia et al. 2014] efficiently reduces Conv to GEMM for plaintext computation, it incurs high masking and rotation costs under FHE due to the expensive process of encoding the encrypted image as a matrix column. Our key innovation, the Ke2Col Conv algorithm, adapts Im2Col for efficient homomorphic execution. Ke2Col enables reuse of the Halevi–Shoup diagonal method [Shai and Victor 2014] to generate a blocked IMRA pattern (Algorithm 3) for Conv, further optimized using Optimize_IMRA from Algorithm 2—the same algorithm used for MVM and other blocked IMRA layouts (Lemma 4.2).

Conv is a key operation in DNN models. An input tensor $I \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ (input channels C_{in}, height H, width W) is convolved with kernels $F \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K_1 \times K_2}$ to produce an output tensor $O \in \mathbb{R}^{C_{\text{out}} \times H_o \times W_o}$. The output dimensions are $H_o = \lfloor (H+2P_1-K_1)/S_1 \rfloor + 1$ and $W_o = \lfloor (W+2P_2-K_2)/S_2 \rfloor + 1$, where ($S_1$ and S_2) (strides) and P_1 and P_2 *padding* control the output size.

For each output channel co, the kernel $F[co] \in \mathbb{R}^{C_{\text{in}} \times K_1 \times K_2}$ slides across the input tensor I, computing a local dot product at each spatial position (h, w) in the output:

$$\mathbf{O}[co, h, w] = conv(I, F) = \sum_{ci=0}^{C_{\text{in}}-1} \sum_{i=0}^{K_1-1} \sum_{j=0}^{K_2-1} F[co, ci, i, j] \times I[ci, h+i-P_1, w+j-P_2] \tag{12}$$

where $\boldsymbol{I[ci, h+i-P_1, w+j-P_2]}$ **is treated as zero if** $\boldsymbol{(h+i-P_1) \notin [0, H)}$ **or** $\boldsymbol{(w+j-P_2) \notin [0, W)}$.

We assume Conv is *channel-divisible* ($C_{\text{out}} \,\%\, C_{\text{in}} = 0 \vee C_{\text{in}} \,\%\, C_{\text{out}} = 0$); otherwise, I and F are zero-padded to satisfy this. We further assume each row of I fits into $\mathcal{S}$, which is reasonable for practical PPML applications since $\mathcal{S} \geqslant 32\text{K}$ [Cheon et al. 2017; Samardzic et al. 2021, 2022].

For a tensor T of shape $[m_0, m_1, \ldots, m_{r-1}]$, $T[m_0][m_1]\ldots[m_{s-1}]$ or $T[m_0, m_1, \ldots, m_{s-1}]$ refers to the subtensor obtained by fixing the first s indices and varying the remaining $r-s$ indices.

We present our approach for a Conv kernel with $K_1 = K_2 = K$, stride $S_1 = S_2 = S = 1$, and padding $P_1 = P_2 = P = (K-1)/2$, preserving the input's spatial dimensions ($H_o = H, W_o = W$)—the most common scenario in practice. We assume I and O can each be stored in a single ciphertext. Discussion of other cases is deferred (Section 4.2.4). Our motivating example is shown in Figure 6.

4.2.1 The Classic Im2Col Algorithm. A naive homomorphic Conv approach is to reuse the Im2Col (image-to-column) method [Jia et al. 2014], widely used for plaintext convolutions. Im2Col reshapes the input tensor I of shape $[C_{\text{in}}, H, W]$ into a matrix I_{col} of shape $[C_{\text{in}} \times K \times K, H \times W]$, where each column represents a flattened receptive field to be convoluted with the kernel. The kernel F of shape $[C_{\text{out}}, C_{\text{in}}, K, K]$ is reshaped into a matrix F_r of shape $[C_{\text{out}}, C_{\text{in}} \times K \times K]$, with each row representing a flattened filter. The convolution reduces to a GEMM between F_r and I_{col}:

$$O_{\text{Im2Col}} = F_r \times I_{\text{col}} \tag{13}$$

where each row of O represents an output channel. The Im2Col transformation is defined as:

$$F_r[co, ci \times K \times K + i \times K + j] = F[co, ci, i, j] \tag{14}$$

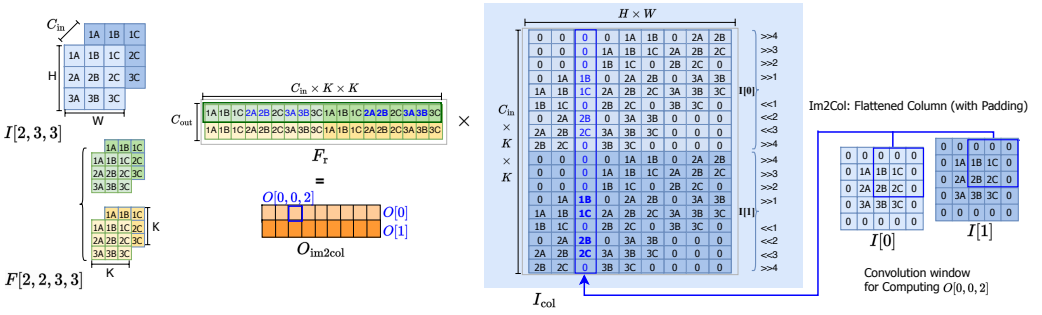


Fig. 6. The classic Im2Col algorithm for convolution: (a) an illustrative example; (b) the Im2Col transformation.

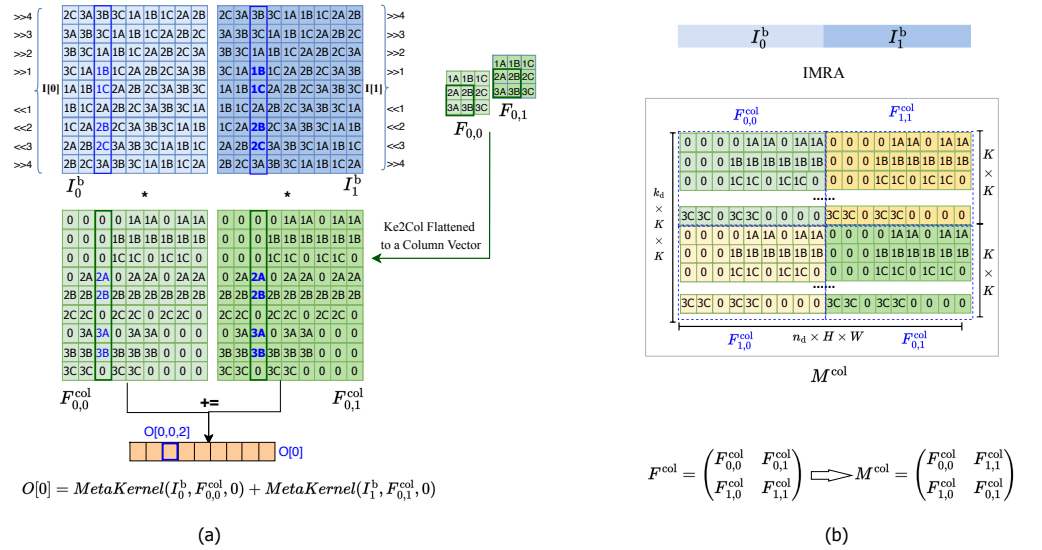


Fig. 7. The Ke2Col algorithm, inspired by Im2Col, for homomorphic convolution: (a) the Ke2Col transformation; (b) an FHE kernel using a MetaKernel-based IMRA pattern enabled by Ke2Col.

$$I_{\text{col}}[ci \times K \times K + i \times K + j, h \times W + w] = I[ci, h + i - P, w + j - P] \tag{15}$$

where $I[ci, h+i-P, w+j-P]$ is treated as zero padding if $(h+i-P) \notin [0, H)$ or $(w+j-P) \notin [0, W)$.

Figure 6 illustrates Im2Col: Figure 6(b) builds on the example in Figure 6(a), showing how $O[0, 0, 2]$ is computed from the third column of I_{col}, which flattens the input convolution window.

While Im2Col enables Conv via optimized GEMM for plaintext computation, it is unsuitable for FHE. Applying Im2Col to encrypted input I requires expensive masking and rotations.

4.2.2 The New Ke2Col Algorithm. We present Ke2Col, a novel FHE-optimized algorithm inspired by Im2Col. Unlike Im2Col, which transforms the input I, Ke2Col transforms the kernel F, preserving the output tensor O's contiguous CHW layout as ciphertext. This approach achieves superior performance in homomorphic Conv, enabling efficient expression via MetaKernels (Equation (10)).

Let $I[ci]$ be the ci-th channel of I in HW (row-major) format. From Im2Col (Equation (15)), we observe that the $(i \times K + j)$-th row in $I_{\text{col}}[ci]$ is a cyclic shift of $I[ci]$ by $Algn[i \times K + j]$, defined as:

$$Algn = [(i - P) \times W + (j - P)]_{i=0,j=0}^{K-1,K-1} \tag{16}$$

Given $Algn$ (a one-dimensional array of rotation offsets), I_{col} in Equation (15) can be obtained as:

$$I_{\text{col}}[ci \times K \times K + i \times K + j] = \overline{\text{HRot}}(I[ci], Algn[i \times K + j]) \tag{17}$$

Here, we apply the same constraint for $I[ci, h, w]$: $I[ci, h + i - P_1, w + j - P_2]$ is treated as 0 if $(h + i - P_1) \notin [0, H)$ or $(w + j - P_2) \notin [0, W)$. This transformation is illustrated in Figure 6(b).

Based on this, we introduce Ke2Col (*kernel to column*), which performs Conv by organizing kernel weights into a column-oriented matrix. We derive I^{b}_{ci} (size $(K \times K) \times (H \times W)$) from $I[ci]$ by cyclically shifting its i-th row by $Algn[i]$ using $\overline{\text{HRot}}$ (VReplicate defined in earlier):

$$I^{\text{b}}_{ci} = \text{VReplicate}(I[ci], K \times K, Algn) \tag{18}$$

Next, as described in Section 4.2.3, we transform the kernel tensor $F \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K \times K}$ in Equation (12) into a column-oriented matrix F^{col} to align with the input structure in Im2Col. For each kernel $F[co, ci] \in \mathbb{R}^{K \times K}$, F^{col} contains a submatrix $F^{\text{col}}_{co,ci} \in \mathbb{R}^{(K \times K) \times (H \times W)}$. The $(h \times W + w)$-th column of $F^{\text{col}}_{co,ci}$ (used to compute $O[co, h, w]$) aligns with I^{b}_{ci}, with padded positions set to zero:

$$F^{\text{col}}_{co,ci}[i \times K + j, h \times W + w] = \begin{cases} F[co, ci, i, j] & \text{if } (h + i - P) \in [0, H) \wedge (w + j - P) \in [0, W) \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

Now, the computation for output channel co follows a MetaKernel-based pattern:

$$\begin{aligned} O[co] &= \text{HAddCC}^{C_{\text{in}-1}}_{ci=0} \text{MetaKernel}(I^{\text{b}}_{ci}, F^{\text{col}}_{co,ci}, 0) \\ &= \text{HAddCC}^{C_{\text{in}-1}}_{ci=0} \overline{\text{HRot}}(\text{HMulCP}^{K \times K - 1}_{i=0}(I^{\text{b}}_{ci}[i], F^{\text{col}}_{co,ci}[i]), 0) \end{aligned} \tag{20}$$

where $\overline{\text{HRot}}$ is currently unnecessary but will be required as the approach develops.

Figure 7 illustrates Ke2Col. In Figure 7(a), Ke2Col computes $O[0]$ using two such MetaKernels for Figure 6(a), highlighting the use of the third columns of I^{b}_0, I^{b}_1, $F^{\text{col}}_{0,0}$, and $F^{\text{col}}_{0,1}$ to compute $O[0, 0, 2]$. It also shows $Algn = [-4, -3, -2, -1, 0, 1, 2, 3, 4]$ used to derive I^{b}_0 from $I[0]$ and I^{b}_1 from $I[1]$.

4.2.3 Generating Optimal FHE-based Conv Kernels. We formalize MKR for $O = conv(I, F)$, assuming both I and O fit in one ciphertext. Tensor partitioning is addressed in Section 4.2.4. A key innovation is to derive a blocked IMRA pattern by interpreting it as an MVM, applying the Halevi-Shoup diagonal method [Shai and Victor 2014] (Algorithm 3), and using Algorithm 2 to further optimize it, particularly for small inputs like CIFAR-10 ($3 \times 32 \times 32$) [He et al. 2015]. Our approach ensures the tensor output O matches the layout of its unencrypted counterpart (in CHW format).

To introduce Algorithm 3, we mimic the notation used for MVM in Algorithm 1, assuming I has k input channels and O has n output channels. Constructing $n \times k$ matrices $F^{\text{col}}_{co,ci}$ from F (line 1) using Equation (19), we create the following column-oriented matrix for F (line 2):

$$F^{\text{col}} = \begin{bmatrix} F^{\text{col}}_{0,0} & F^{\text{col}}_{0,1} & \cdots & F^{\text{col}}_{0,k-1} \\ F^{\text{col}}_{1,0} & F^{\text{col}}_{1,1} & \cdots & F^{\text{col}}_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ F^{\text{col}}_{n-1,0} & F^{\text{col}}_{n-1,1} & \cdots & F^{\text{col}}_{n-1,k-1} \end{bmatrix} \in \left(\mathbb{R}^{(K \times K) \times (H \times W)}\right)^{n \times k} \tag{21}$$

In line 4, we obtain I^{b} from I. Treating I^{b} as a vector of k channels, its ci-th element is I^{b}_{ci} (defined in Equation (18)). The convolution $O = conv(I, F)$ can now be computed as an MVM: $O = F^{\text{col}} \times I^{\text{b}}$, where element-wise multiplication and addition are defined in Equation (20): $F^{\text{col}}_{co,ci} \times I^{\text{b}}_{ci} = \text{HMulCP}^{K \times K - 1}_{i=0}(I^{\text{b}}_{ci}[i], F^{\text{col}}_{co,ci}[i])$ and $I^{\text{b}}_{ci} \times F^{\text{col}}_{co,ci} + I^{\text{b}}_{ci'} \times F^{\text{col}}_{co,ci'} = \text{HAddCC}(I^{\text{b}}_{ci} \times F^{\text{col}}_{co,ci}, I^{\text{b}}_{ci'} \times F^{\text{col}}_{co,ci'})$.

Algorithm 3: Code generation for Conv with slot size $\mathcal{S}$.

Input: $I \in \mathbb{R}^{1\times(k\times H\times W)}$ (in ciphertext) in CHW layout and kernel $F \in \mathbb{R}^{n\times k\times K\times K}$

Output: FHE program for $O = conv(I, F)$ with $O \in \mathbb{R}^{1\times(n\times H\times W)}$ in CHW layout

1 $\forall\ 0 \leqslant co < n \wedge 0 \leqslant ci < k$: Construct $F^{\text{col}}_{co,ci} \in \mathbb{R}^{(K\cdot K)\times(H\cdot W)}$ for F (Equation (19)) ;

2 Construct $F^{\text{col}} \in \left(\mathbb{R}^{(K\times K)\times(H\times W)}\right)^{n\times k}$ from $F^{\text{col}}_{co,ci}$ (Equation (21)) ;

3 $M^{\text{col}} \in \mathbb{R}^{\left(n_d\cdot K\cdot K\right)\times\left(k_d\times H\times W\right)} \leftarrow$ IMRAPacking(F^{col}), where $n_d = \min(n,k)$ and $k_d = \max(n,k)$;

4 $I^{\text{b}} \in \mathbb{R}^{(K\times K)\times(k\times H\times W)}$ = VReplicate$(I, K\times K, Algn)$ with $Algn$ from Equation (16);

5 $I^{\text{b}}_{\text{rep}} \in \mathbb{R}^{(K\times K)\times(\frac{k_d}{n_d}\times k\times H\times W)} \leftarrow$ HReplicate$(I_{\text{b}}, \frac{k_d}{k})$;

6 $OptPat_{\text{IMRA}} \leftarrow$ Optimize_IMRA ("O_{IMRA} = IMRA$(I^{\text{b}}_{\text{rep}}, M^{\text{col}}, Shift = H\times W)$");

7 Generate the following FHE-based kernel code:

8
```
OptPat_IMRA (with O_IMRA containing the result) ;   // 8a
O_IMRA ← Reduce(O_IMRA, k/n, k × H × W) ;           // 8b
O ← HMulCP(O_IMRA, mask[1]_{n×H×W}) ;               // 8d
```

Revisiting Figure 7(a), $F^{\text{col}} = [[F^{\text{col}}_{0,0}, F^{\text{col}}_{0,1}]; [F^{\text{col}}_{1,0}, F^{\text{col}}_{1,1}]]$ is constructed for the example in Figure 6(a). Given I^{b}_0, I^{b}_1, $F^{\text{col}}_{0,0}$, and $F^{\text{col}}_{0,1}$ displayed explicitly, we compute $O[0] = F^{\text{col}}_{0,0} \times I^{\text{b}}_0 + F^{\text{col}}_{0,1} \times I^{\text{b}}_1$, where $\times$ and $+$ are homomorphic multiplication and addition defined above.

We apply the Halevi-Shoup diagonal method [Shai and Victor 2014] to the MVM formulation $O = F^{\text{col}} \times I^{\text{b}}$, creating a rotation-efficient layout for F^{col} by constructing $M^{\text{col}} =$ IMRAPacking$(F^{\text{col}}) = \mathscr{M}(\mathscr{D}(F^{\text{col}}))$ (line 3), flattening its individual sub-matrix elements, where $\mathscr{D}$ and $\mathscr{M}$ are defined in Equation (5) and Equation (7). We also create $I^{\text{b}}_{\text{rep}}$ from I^{b} by replication (if needed) for dimension compatibility with tall matrices F^{col} (line 5), as in Algorithm 1 (line 2) and shown in Figure 3(b).

We derive a blocked IMRA pattern (line 6), optimized using Optimize_IMRA from Algorithm 2 to produce the FHE-based Conv kernel (line 8), where line 8b mirrors line 5b in Algorithm 1 for short matrices, requiring reduction to combine partial convolution results across input channels. In line 6, $Shift = H\times W$ is used since adjacent channels in I are separated by $H\times W$ (in CHW layout).

Applying Algorithm 3 to the example in Figure 6(a) with $\mathcal{S} = 18$, we set $I^{\text{b}}_{\text{rep}} = I^{\text{b}} = [I^{\text{b}}_0, I^{\text{b}}_1]$ and $F^{\text{col}} = [[F^{\text{col}}_{0,0}, F^{\text{col}}_{0,1}]; [F^{\text{col}}_{1,0}, F^{\text{col}}_{1,1}]]$ (from Figure 7(a)) to compute M^{col} (Figure 7(b)). Since $\mathcal{S} = 18$, Optimize_IMRA (Algorithm 2) returns the input pattern without further optimization (line 6):

$$O_{\text{IMRA}} = \text{HAddCC}(\text{MetaKernel}([I^{\text{b}}_0, I^{\text{b}}_1], [F^{\text{col}}_{0,0}, F^{\text{col}}_{1,1}], 0), \text{MetaKernel}([I^{\text{b}}_0, I^{\text{b}}_1], [F^{\text{col}}_{1,0}, F^{\text{col}}_{0,1}], 9)) \quad (22)$$

whose homomorphic execution is illustrated in Figure 8. Here, $O = O_{\text{IMRA}}$ contains the two output channels in CHW layout for Figure 6(a), ensuring compatibility with the expected output format.

Theorem 4.4. *The FHE program generated by Algorithm 3 for Conv is correct.*

Proof. Correctness follows from: (1) Ke2Col maintains consistency with Im2Col, (2) the IMRA pattern in Algorithm 3 (line 6) remains consistent with Ke2Col via Halevi-Shoup diagonalization [Shai and Victor 2014], and (3) optimization by Algorithm 2 preserves correctness (Lemma 4.2). □

4.2.4 Discussion. Although presented for square filters $K\times K$, our MKR approach naturally extends to non-square filters since Ke2Col's foundation (Equations (16) to (19)) is size-independent.

If the input I or output O tensor (Equation (12)) exceeds ciphertext capacity, we partition them along the channel dimension. Given input shape $[C_{\text{in}}, H, W]$, I is split into $I^P = \{I^P_0, I^P_1, \ldots, I^P_{P_I-1}\}$, where each I^P_j (with $C_{\text{in},j}$ channels) fits into one ciphertext. Likewise, O is partitioned into $O^P =$

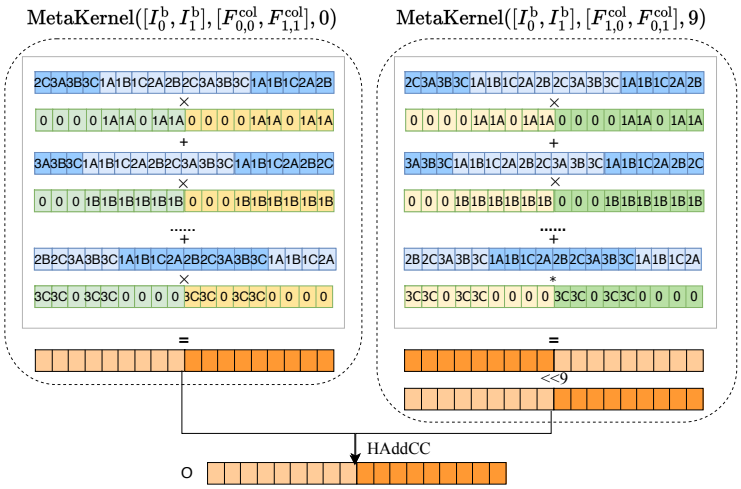


Fig. 8. Homomorphic execution of the FHE program specified in Equation (22), as derived in Figure 7.

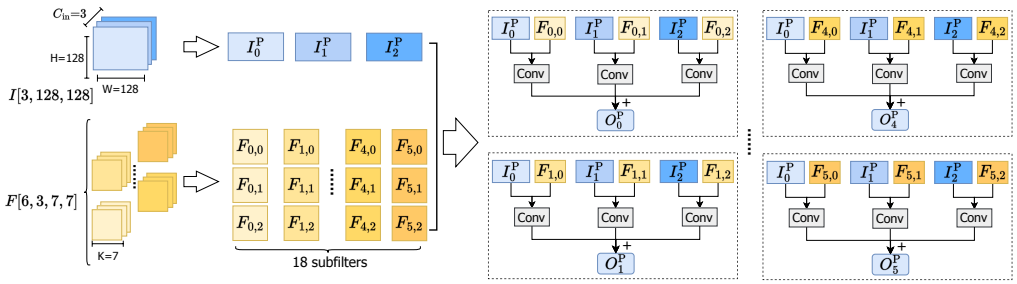


Fig. 9. MKR's partitioning of $O = conv(I, F)$ under $\mathcal{S}$ = 32K, where each input and output channel fits in a ciphertext. Input $I[3, 128, 128]$ is split into $\{I_0^P, I_1^P, I_2^P\}$, and output $O[6, 128, 128]$ into $\{O_0^P, \ldots, O_5^P\}$, each of shape $[1, 128, 128]$. Filter $F[6, 3, 7, 7]$ is divided into 18 smaller filters $F_{i,j}$ of shape $[1, 1, 7, 7]$, one per channel.

$\{O_0^P, O_1^P, \ldots, O_{P_O-1}^P\}$, with each O_i^P holding $C_{\text{out},i}$ channels. The kernel F is split into P_I and P_O parts along input and output channels, with $F_{i,j} \in \mathbb{R}^{C_{\text{out},i} \times C_{\text{in},j} \times K \times K}$. Let $O_{i,j}^P = conv(I_j^P, F_{i,j})$ be the partial output computed via Optimize_IMRA (Algorithm 3); the full output is $O_i^P = \sum_{j=0}^{P_I} O_{i,j}^P$. As shown in Section 6, MKR yields lower multiplicative depth than Fhelipe for large Conv kernels.

When a channel exceeds ciphertext capacity ($H \times W > \mathcal{S}$), we segment it along H (given $W < S$). Each segment (including halo) must fit within one ciphertext, following the same formalism.

Figure 9 shows MKR's optimization of a Conv kernel where each input and output channel of a $3 \times 128 \times 128$ image fits into a single ciphertext under $\mathcal{S}$ = 32K. For larger inputs like ImageNet ($3 \times 224 \times 224$), channels are partitioned so that half a channel fits in one ciphertext.

Our MetaKernel-based IMRA pattern naturally supports depthwise convolution in models like MobileNet [Howard et al. 2017], where each input channel is convolved with its own kernel. We map the filter $F \in \mathbb{R}^{C_{\text{in}} \times K \times K}$ to $F^{\text{col}} = [F_0^{\text{col}}, \ldots, F_{C_{\text{in}}-1}^{\text{col}}]$ using channel-aligned placement with input I. A single MetaKernel suffices, as depthwise convolution avoids cross-channel interaction.

Non-unity strides create gaps in the output tensor, requiring a compaction step to extract valid elements using masking and rotations, as in Fhelipe [Krastev et al. 2024].

4.3 Analysis

We now compare MKR with FHE-MP-CNN [Lee et al. 2022b] and Fhelipe [Krastev et al. 2024].

For MVM $C = B \times A$, as motivated in Section 3, MKR can significantly reduce total rotations compared to both, and unlike Fhelipe, preserves the unencrypted output layout.

For Conv $O = conv(I, F)$, the Single-Input Single-Output (SISO) approach—first introduced in Gazelle [Juvekar et al. 2018]—is widely used in FHE-based DNNs. FHE-MP-CNN and Fhelipe extend SISO to multi-channel convolutions by applying it independently to each input-output channel pair and aggregating the results. This produces non-contiguous outputs, requiring costly HRot operations to restore the expected layout. In contrast, MKR adopts a different tensor partitioning strategy that preserves layout compatibility (Figure 9) and computes each smaller Conv kernel using a new Ke2Col-enabled, MetaKernel-based approach (Figure 7).

Table 2. Rotations and slot utilization for MVM and Conv.

Method	MVM		Conv	
	#Rotations	Slot Utilization	#Rotations	Slot Utilization
FHE-MP-CNN	$n+k-2$	$\frac{n \times k}{S \times (n+k-1)}$	$(K \times K - 1) + \left\lceil \frac{C_{\text{out}}}{2^{\lfloor \log_2(\frac{S}{HWC_{\text{in}}}) \rfloor}} \right\rceil (\lceil \log_2 C_{\text{in}} \rceil) + C_{\text{out}} + \log_2 2^{\lfloor \log_2(\frac{S}{HWC_{\text{out}}}) \rfloor}$	100%
Fhelipe	$\log_2\left(\frac{\min(S, n \times k)}{k}\right) + \log_2(k) \times \left\lceil \frac{n \times k}{S} \right\rceil + \left\lceil \frac{n \times k}{S} \right\rceil - 1$	$\frac{n \times k}{S \times \left\lceil \frac{n \times k}{S} \right\rceil}$	$(K \times K - 1) + (K \times K) \times \log_2\left(\frac{S}{C_{\text{in}} \times H \times W}\right) + \frac{C_{\text{out}} \times C_{\text{in}} \times H \times W}{S} \times \log_2 C_{\text{in}} + \left(\frac{C_{\text{out}} \times C_{\text{in}} \times H \times W}{S} - 1\right)$	100%
MKR	Per Cost Model (Equation (11))			

Table 2 compares MKR, FHE-MP-CNN, and Fhelipe on MVM and Conv kernels, reporting total rotations and slot utilization. Slot utilization reflects packing of plaintext weights (B for MVM and F for Conv), affecting alignment with encrypted inputs. Formulas for FHE-MP-CNN and Fhelipe follow their algorithms, except FHE-MP-CNN's Conv rotations, simplified from [Lee et al. 2022b] for $K \times K$ filters with unit stride. FHE-MP-CNN targets only small inputs, designed for ResNet-20 on CIFAR-10. MKR's metrics are from our cost model in Equation (11).

Unlike these two representative methods, MKR minimizes rotations and maximizes slot utilization, significantly advancing the state of the art, as demonstrated in Section 6.

5 Implementation

We implemented MKR in the open-source ANT-ACE compiler [Li et al. 2025], which compiles DNN models into C/C++ FHE programs using CKKS [Cheon et al. 2019, 2017] for CPU execution. MVM and Conv kernels share a unified template, enabling efficient MetaKernel instantiation. Our algorithms are integrated into ANT-ACE's `Vector IR`, which expresses tensor operations as vector operations using ANT-LIB, an FHE library supporting efficient CKKS operations (e.g., HRot, HAddCC, and HMulCP). MKR, with about 3500 lines of C++, will be open-sourced soon (Section 8). We validated our approach through extensive benchmarking of diverse kernels and DNN models.

6 Evaluation

Our objective is to demonstrate that MKR significantly outperforms the state of the art in accelerating DNN inference under the CKKS scheme [Cheon et al. 2019, 2017].

We compare MKR with Fhelipe [Krastev et al. 2024] in terms of DNN inference performance at both the operator (individual MVM and Conv kernels) and model (full DNNs) levels on a single CPU. Fhelipe implements homomorphically MVM as shown in Figure 2, and Conv using the

SISO approach [Juvekar et al. 2018], as described in Section 4.3. This comparison is justified for four reasons: (1) Fhelipe represents state-of-the-art tensor layout transformations, matching or exceeding hand-tuned implementations like FHE-MP-CNN [Lee et al. 2022b]. (2) Unlike FHE-MP-CNN, which supports only single-ciphertext inputs (e.g., ResNet-20 on CIFAR-10), Fhelipe handles various input sizes (Section 2). (3) Since research on FHE compilers remains in its early stages, existing kernel-generation efforts primarily aim to enable homomorphic execution on single CPUs [Dathathri et al. 2020, 2019; Li et al. 2025; Zhang et al. 2025] rather than optimize performance. (4) Like CHET [Dathathri et al. 2019] and EVA [Dathathri et al. 2020], Fhelipe compiles DNNs into static FHE circuits by unrolling all loops (which must have constant bounds in FHE), substantially increasing compile time and preventing compilation of large DNN models, as previously observed [Zhang et al. 2025] and confirmed here. Moreover, the absence of loop structures complicates multi-core execution of FHE circuits. Multi-core parallelization, enabled by MKR's MetaKernel-level parallelism, will be explored in future work.

Therefore, our evaluation addresses the following three research questions:

- **RQ1**: Can MKR outperform Fhelipe across MVM and Conv kernels with varying sizes?
- **RQ2**: Can MKR outperform Fhelipe across DNN models?
- **RQ3**: Does MKR generate correct FHE-based kernels efficiently?

Despite recent advances [Cheon et al. 2017; Gerami et al. 2024; Luo et al. 2023], FHE remains up to 10,000× slower for small machine learning models [Krastev et al. 2024; Li et al. 2025]. To address FHE's slow encrypted inference, this work reduces rotations (and thus inference time) while maintaining high slot utilization for plaintext weights to improve MVM and Conv efficiency. Unlike Fhelipe, which maximizes slot usage via full replication at the cost of excessive rotations, MKR balances rotation overhead and slot utilization using a rotation-aware cost model, enabling efficient support for large DNN models with inputs exceeding single ciphertext capacity.

For a plaintext tensor T of shape $[m_0, m_1, \ldots, m_{r-1}]$, its slot utilization rate is defined as $\prod_{i=0}^{r-1} m_i/(\mathcal{S} \times n_{\text{ct}})$, where n_{ct} is the number of plaintexts used to store T.

All experiments for Fhelipe and MKR were conducted on a single Intel(R) Xeon(R) Platinum 8369B CPU at 2.70 GHz with 1 TB RAM, running Ubuntu 20.04. For fair comparison, both methods used the same CKKS parameters: input scale $\Delta = 2^{56}$, output scale $q_0 = 2^{60}$, ring degree $N = 2^{16}$ (i.e., $\mathcal{S} = N/2 = 32K$) (Section 2.1), and ciphertext modulus Q set according to the multiplicative depth to ensure 128-bit security [Albrecht et al. 2019; Bossuat et al. 2024].

6.1 RQ1: DNN Kernels

We evaluate 10 representative MVM and Conv kernels with small and large inputs (Table 3), including ciphertext input (shape), plaintext weight (shape and size), and FLOPs (floating-point operations performed). The eight Conv kernels include four from ResNet18 with ImageNet inputs (marked "Large"), where each image exceeds a single ciphertext, and four from ResNet20 with CIFAR10 inputs, where each image fits into one ciphertext. For Conv kernels (Equation (12)), "Ciphertext Input" refers to the input tensor I, and "Plaintext Weight" refers to the kernel tensor F with shape $[C_{\text{out}}, C_{\text{in}}, H, W]$ and size $C_{\text{out}} \times C_{\text{in}} \times H \times W$. For MVM kernels ($C = B \times A$), including MVM1 from AlexNet and MVM2 from ImageNet in VGG11, "Ciphertext Input" refers to A with shape $[k]$, and "Plaintext Weight" refers to B with shape $[n, k]$ and size $n \times k$.

As noted in Section 3, aligning plaintext weights with ciphertext inputs is challenging but essential for minimizing costly ciphertext rotations and ensuring efficient homomorphic execution.

For both MVM and Conv, MKR substantially outperforms Fhelipe by significantly reducing rotations, despite Fhelipe achieving better slot utilization. This shows that blindly adopting full data replication does not yield high inference performance, which is essential for PPML adoption.

Table 3. MVM and Conv kernels from real-world DNN models.

Kernel	Ciphertext Input	Plaintext Weight		FLOPs
	Shape	Shape	Size (KB)	(Millions)
Conv1	[3, 32, 32]	[16, 3, 3, 3]	0.43	0.88
Conv2	[16, 32, 32]	[16, 16, 3, 3]	2.30	4.72
Conv3	[32, 16, 16]	[32, 32, 3, 3]	9.22	4.72
Conv4	[64, 8, 8]	[64, 64, 3, 3]	36.86	4.72
Conv1_Large	[64, 56, 56]	[64, 64, 3, 3]	36.86	230.69
Conv2_Large	[128, 28, 28]	[128, 128, 3, 3]	147.46	230.69
Conv3_Large	[256, 14, 14]	[256, 256, 3, 3]	589.82	230.69
Conv4_Large	[512, 7, 7]	[512, 512, 3, 3]	2,359.30	230.69
MVM1	[4096]	[4096, 4096]	16,777.22	33.55
MVM2	[25088]	[4096, 25088]	102,760.45	205.52

6.1.1 Conv. We evaluate four small Conv kernels (`Conv1`–`Conv4`) and four large ones (`Conv1_Large`–`Conv4_Large`). The small kernels serve for testing and benchmarking, while the large ones reflect practical applications. MKR performs significantly better on the large kernels.

For small Conv kernels (Table 4), MKR outperforms Fhelipe by 10.08× to 15.17×. While Fhelipe achieves 100% slot utilization for kernel weights F, it incurs high rotation costs. For instance, `Conv4` requires 48 rotations for 8 independent computations (each needing $\log_2(C_{\text{in}}) = 6$ rotations), 35 for aligning I with F, and 7 for merging ciphertexts in its final layout conversion step, totaling 90. In contrast, MKR requires only 32 rotations—14 for replicating I ($\gamma(P_s + 1) + bs^{\text{opt}} - 1$), 15 for accumulating results from 16 MetaKernels ($P_b/P_s - 1$), and 3 for result reduction ($P_s - 1$), according to its cost model (Equation (11)) with the values of these parameters listed in Table 8. By reducing Fhelipe's excessive rotations and eliminating its final layout conversion, MKR ensures output tensor layout compatibility. Both methods require the same multiplicative depth of 2.

For large Conv kernels (Table 5) used in practical applications, where I, O, and F span multiple ciphertexts, MKR outperforms Fhelipe by 88.21×–153.28×. Both methods achieve 100% slot utilization for F, but Fhelipe incurs significantly higher rotation costs due to extensive masking and cross-ciphertext reconstruction, requiring a multiplicative depth of 4. In contrast, MKR partitions tensors I and O along the channel dimension and aligns F accordingly (Figure 9), distributing subtensors across ciphertexts (as detailed in Section 4.2.4, with partitioning parameters P_I and P_O given in Table 8). This strategy reduces the required multiplicative depth to 2. For example, `Conv2_Large` with $I[128, 28, 28]$ and $F[128, 128, 3, 3]$ requires Fhelipe to replicate I 128 times, using 512 ciphertexts with $\log_2(128) = 7$ rotations each, totaling $512 \times 7 = 3584$ rotations (out of 4142 total). In contrast, MKR 's MetaKernel-based approach reduces this total to only 1060 rotations.

6.1.2 MVM. Table 6 shows results for two MVM kernels. MKR outperforms Fhelipe by generating far fewer rotations, though it has lower slot utilization for B in `MVM1`.

Let us take `MVM1` ($B[4096, 4096]$) as an example. Fhelipe requires 512 independent computations to produce $C[4096]$, with each handling $(4096 \times 4096)/\mathcal{S} = 8$ elements of C. Each group of 8 elements is computed via ciphertext-level summation across $\log_2(4096) = 12$ rotations, totaling $512 \times 12 = 6144$ rotations. The 512 ciphertexts hold partial results for C, requiring 511 additional rotations to merge them into a single output (Figure 2). In contrast, MKR uses an IMRA pattern layout with optimal parameters $P_b = 128$ and $P_s = 4$ with $bs^{\text{opt}} = 32$ inferred (Table 8), requiring only $P_b/P_s = 32$ MetaKernels. This enables all 4096 elements of C to be computed within a single ciphertext, eliminating intermediate rotations and post-computation merging. For large MVM kernels, Fhelipe's full replication strategy for B becomes increasingly inefficient as B grows.

For `MVM2`, the Fhelipe-generated version fails to finish within 20 hours, while MKR completes in just 49.03 seconds due to excessive rotations generated with Fhelipe's 65,535 vs. MKR's 135.

Table 4. Comparison of MKR and Fhelipe for small Conv Kernels.

Kernel	Fhelipe			MKR			
	Slot Utilization	Rotations	Time (secs)	Slot Utilization	Rotations	Time (secs)	Speedup
Conv1	100%	40	29.45	50%	17	2.26	13.03×
Conv2	100%	56	42.25	50%	26	4.19	10.08×
Conv3	100%	73	43.85	50%	28	2.89	15.17×
Conv4	100%	90	45.01	50%	32	3.12	14.43×

Table 5. Comparison of MKR and Fhelipe for large Conv Kernels.

Kernel	Fhelipe			MKR			
	Slot Utilization	Rotations	Time (secs)	Slot Utilization	Rotations	Time (secs)	Speedup
Conv1_Large	100%	3,220	12,089	100%	1,096	88.50	136.60×
Conv2_Large	100%	4,142	13,686	100%	1,060	89.29	153.28×
Conv3_Large	100%	4,628	13,508	100%	1,042	101.01	133.73×
Conv4_Large	100%	5,127	13,220	100%	1,033	149.87	88.21×

Table 6. Comparison of MKR, Fhelipe, and BSGS for MVM kernels.

Kernel	Fhelipe			BSGS			MKR				
	Slot Utilization (B)	Rotations	Time (secs)	Slot Utilization (B)	Rotations	Time (secs)	Slot Utilization (B)	Rotations	Time (secs)	Speedup (Fhelipe)	Speedup (BSGS)
MVM1	100.00%	6,658	3,019.95	12.50%	193	55.58	50.00%	71	16.13	187.23×	3.45×
MVM2	76.56%	65,535	>20h	76.56%	384	343.71	76.56%	135	49.03	–	7.01×

In Table 6, we also compare MKR with the Baby-Step Giant-Step (BSGS) algorithm from HELIB [Halevi and Shoup 2018]. BSGS decomposes $C = B \times A$, where B is an $n \times n$ square matrix, into n_1 groups of n_2 diagonals, with $n = n_1 \times n_2$. MKR achieves speedups of 3.45× on MVM1 and 7.01× on MVM2 over BSGS by significantly reducing costly ciphertext rotations. BSGS is primarily used for square matrices in bootstrapping in HELIB, SEAL [SEAL 2020], and OpenFHE [Ahmad et al. 2022], as it is less efficient for non-square matrices. In contrast, MKR supports efficient execution of both MVM and Conv operations with arbitrary input dimensions.

6.1.3 Discussion. MKR achieves significant speedups over Fhelipe across these kernels by dramatically reducing rotations. This optimization directly lowers rotation costs and substantially reduces associated overheads, such as fewer temporary ciphertexts/plaintexts, fewer masking and HMulCP operations required to manage these temporaries, and improved memory efficiency—especially beneficial for large tensor operations common in practical DNN models.

6.2 RQ2: DNN Models

We evaluate a range of deep learning models—ResNet, AlexNet, VGG16, SqueezeNet, and MobileNet—on CIFAR-10 ([3, 32, 32]) and ImageNet ([3, 224, 224]). These models are more complex than those used in prior studies such as Fhelipe [Krastev et al. 2024] and FHE-MP-CNN [Lee et al. 2022b], which exclude large inputs that span multiple ciphertexts. Our evaluation also includes depthwise Conv kernels, which are essential in modern architectures like MobileNet.

Given $\mathcal{S} = 32K$, MKR partitions each ImageNet input into six ciphertexts (half a channel per ciphertext), as detailed in Section 4.2.4. In contrast, Fhelipe partitions inputs along width W, padding dimensions to powers of 2, which leads to excessive rotations and associated overheads.

Table 7 reports average inference times (three runs), detailing MVM and Conv contributions. Like Fhelipe [Krastev et al. 2024], ANT-ACE (which hosts MKR) employs high-degree polynomial approximations for non-linear operations [Lee et al. 2021]. Fhelipe successfully compiled four CIFAR-10 models (single-ciphertext inputs), but failed to handle: (1) MobileNets (due to unsupported depthwise convolutions) and (2) larger models (> 10 hours due to inefficient FHE circuit representations [Zhang et al. 2025]). While Fhelipe compiles the first Conv layer for these larger

Table 7. Comparison of compile and inference times for DNN models generated by MKR and Fhelipe.

Models	Fhelipe (secs)				MKR (secs)				Speedup
	Compile-Time	Runtime			Compile-Time	Runtime			
		Conv	MVM	Total		Conv	MVM	Total	
CIFAR10 Models									
ResNet20	39.50	457.64	0.758	1905.05	1.42	107.05	1.23	1070.37	1.78×
SqueezeNet	39.07	1301.67	0.63	2377.25	2.11	240.92	3.17	1359.2	1.75×
AlexNet	283.55	7950.07	3164.95	11794.99	77.39	78.38	55.72	995.89	11.84×
VGG11	53.69	3965.14	0.80	4,687.14	1.88	66.82	4.91	757.19	6.19×
MobileNet	–	–	–	–	0.97	101.52	10.22	1662.27	–
ImageNet Models									
ResNet18	>10h	OOM (1st Layer)	–	–	49.56	5439.06	4.04	11717	–
SqueezeNet	>10h	OOM (1st Layer)	–	–	18.93	4441.22	4.29	13534	–
AlexNet	>10h	OOM (1st Layer)	–	–	338.21	2049.48	48.38	4364	–
VGG11	>10h	OOM (1st Layer)	–	–	812.14	9275.19	69.14	28652	–
MobileNet	–	–	–	–	12.13	5079.21	4.54	15,594	–

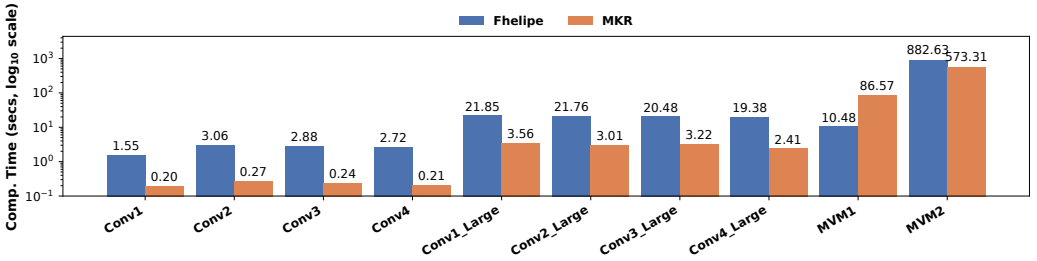


Fig. 10. Comparison of MKR and Fhelipe in compile time for MVM and Conv Kernels.

models, execution resulted in out-of-memory (OOM) errors within 12 hours (on a 1TB system), primarily due to excessive rotation operations and associated computational overheads.

In contrast, MKR compiled and ran all models successfully. For the four Fhelipe-compiled models that ran, MKR achieves 1.75×–11.84× speedups. As input size increases, Conv accounts for 31.80–46.96% of inference time, compared to 10% for small models [Krastev et al. 2024]. These results not only demonstrate MKR's ability to enable the homomorphic execution of large DNNs but also highlight the growing need to accelerate homomorphic Conv performance.

6.3 RQ3: Efficiency and Effectiveness

Compile Times. Figure 10 compares the compile times of MKR and Fhelipe (in $\log_{10}$ scale) across 10 MVM and Conv kernels from Table 3. For MVM kernels, MKR spends most of its time generating the rotation-efficient matrix M_{b} (lines 4–5, Algorithm 2) from the plaintext matrix B, which is typically larger than the Conv kernel matrix F^{col} (Equation (21)). For full DNN models (Table 7), MKR compiles all within 14 minutes. In contrast, Fhelipe takes an order of magnitude longer to compile the first four small models (which completed successfully) and fails to compile the four larger models (labeled ">10h") within 10 hours, due to its reliance on static FHE circuits—which scale poorly for large programs, as previously observed [Zhang et al. 2025] and confirmed here.

Impact of MKR's MVM and Conv Optimization on Model's Total Inference Time. For the four models that run successfully under Fhelipe (Table 7), Figure 11 shows that MKR's speedups over Fhelipe result from significant reductions in Conv and MVM execution times across all models. For example, for `AlexNet-CIFAR10`, which includes 5 Conv and 3 MVM layers across 8 layers, MKR achieves its largest speedup of 11.84× by reducing the total MVM time from 3164.95 seconds to 55.72 (56.80×) and the total Conv time from 7950.07 seconds to 78.38 (101.43×).

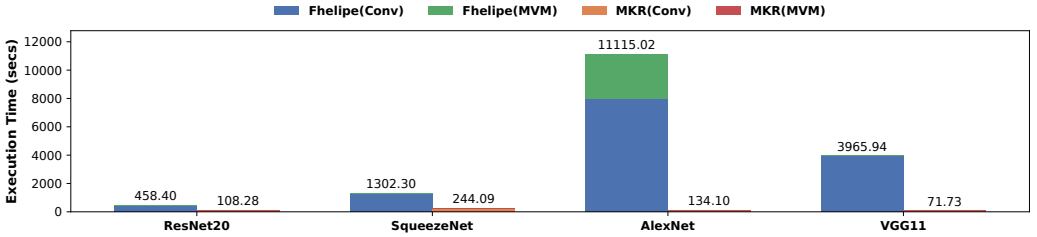


Fig. 11. Comparison of MKR and Fhelipe for Conv and MVM contributions to inference time across models.

Table 8. MKR's tensor partitioning and optimal IMRA blocking results for MVM and Conv kernels.

Kernel	Tensor Partitioning		Optimal Blocking			
	P_I	P_O	Search Space	P_b	P_s	bs^{opt}
Conv1	1	1	3	4	1	9
Conv2	1	1	5	16	1	9
Conv3	1	1	11	32	2	9
Conv4	1	1	18	64	4	9
Conv1_Large	8	8	1	8	1	9
Conv2_Large	4	4	1	32	1	9
Conv3_Large	2	2	1	128	1	9
Conv4_Large	1	1	1	512	1	9
MVM1	n/a	n/a	36	128	4	32
MVM2	n/a	n/a	13	64	1	64

MKR's Tensor Partitioning and Optimal IMRA Blocking. Table 8 summarizes the tensor partitioning and optimal blocking parameters for the 10 kernels listed in Table 3. Tensor partitioning, defined by input/output channel splits (P_I, P_O), follows the scheme described in Section 4.2.4. Optimal blocking, obtained by solving Equation (11), includes the search space size, selected parameters (P_b, P_s), and the resulting block size bs^{opt} (computed in line 2 of Algorithm 2). The reported results indicate that both steps incur negligible compile-time overhead.

Correctness. Given the slow performance of FHE, we validated MKR's correctness using the 10 kernels in Table 3. For each kernel optimized by MKR, encrypted results were decrypted and compared with unencrypted results. MKR maintained a relative error threshold of 0.001% (1×10^{-5}), ensuring three significant digits of precision and validating correctness at the operator level.

Inference Accuracy. Given FHE's slow DNN inference speeds (Table 7), inference accuracy is typically validated using small DNN models and a limited set of image samples. Fhelipe reports less than 1.0% accuracy loss for ResNet-20, RNN, and LogReg [Krastev et al. 2024], while ANT-ACE also averages under 1.0%. Using 1,000 CIFAR-10 images per model, MKR, implemented in ANT-ACE, maintains the same average accuracy for all models. In comparison, expert hand-tuned implementations [Lee et al. 2022b] also show an accuracy drop of 0.1% to 0.6%.

7 Related Work

Recent advances in CKKS [Cheon et al. 2019, 2017] have improved the practicality of PPML [Gürsoy et al. 2020] by enabling SIMD parallelism and fixed-point arithmetic. FHE Libraries like SEAL [SEAL 2020] and OpenFHE [Ahmad et al. 2022] provide fast FHE operations and simplify development. However, building efficient, correct, and secure FHE applications remains challenging and error-prone, often taking weeks even for experts [Krastev et al. 2024].

To ease development, several FHE compilers have emerged—nGraph-HE [Boemer et al. 2019], HECATE [Yongwoo et al. 2022], DECAPO [Cheon et al. 2024], CHET [Dathathri et al. 2019], EVA [Dathathri et al. 2020], Fhelipe [Krastev et al. 2024], and ANT-ACE [Li et al. 2025]—primarily targeting small DNNs with CIFAR inputs. These compilers focus on managing rescaling [Cheon et al. 2024; Dathathri et al. 2020; Liu et al. 2025; Yongwoo et al. 2022] and bootstrapping [Cheon et al. 2024; Krastev et al. 2024; Liu et al. 2025], often with DSL support. Recent efforts have further advanced end-to-end FHE compilation pipelines. Orion [Ebel et al. 2025] translates PyTorch-based DNNs into FHE programs and supports high-resolution tasks such as ImageNet classification and YOLO-v1 object detection. Cinnamon [Jayashankar et al. 2025], by contrast, targets system-level scalability, proposing a scale-out accelerator architecture and parallel FHE compilation strategies for large models like BERT, yielding notable speedups over CPU execution.

As reviewed in Section 2, layout optimization for large tensor operations in FHE remains nascent, with FHE-MP-CNN [Lee et al. 2022b] (hand-tuned) and Fhelipe [Krastev et al. 2024] (DSL-based) representing the state of the art. FHE-MP-CNN provides optimized kernels for ResNet, while Fhelipe automates layout transformations. MKR introduces MetaKernel-based compilation, outperforming Fhelipe on MVM/Conv and integrating seamlessly with the open-source FHE compiler ANT-ACE.

Fhelipe [Krastev et al. 2024] was the first FHE compiler to automate MVM and Conv translation for arbitrary input sizes. However, its full-replication strategy for plaintext weights leads to excessive rotations, significantly impacting performance. Moreover, its loop-free FHE circuit representation harms compile-time scalability for large DNN models. In contrast, MKR employs a rotation-aware cost model to co-optimize rotation costs and slot utilization, achieving superior performance. Implemented in ANT-ACE, which preserves loop structures in its IR, MKR demonstrates both compile-time efficiency and scalability when compiling large DNN models.

Prior work uses microkernels for efficient plaintext computation: POCA [Su et al. 2017] generates vectorized GEMM kernels, OneDNN [Jianhui Li and Lavery 2024] employs batch-reduce GEMM, and MikPoly [Yu et al. 2024] combines fixed-size kernels dynamically. Unlike these, MKR introduces MetaKernels that compose horizontally (enhancing SIMD parallelism) and vertically (improving computational parallelism) for high-performance homomorphic MVM and Conv operations.

8 Conclusion

We introduce MKR, a new compiler approach for generating high-performance homomorphic MVM and Conv kernels in a unified framework. MKR explores MetaKernel-based IMRA layouts to minimize rotations while maintaining high slot utilization. It advances the state of the art with a rotation-aware cost model that balances rotation overhead and slot utilization, handles arbitrary input sizes, preserves output tensor layout, and significantly outperforms prior work.

We plan to extend our work in several directions. First, with the growing importance of privacy in LLMs, we aim to integrate MKR with batch matrix-matrix product, a key operator in inference, to enhance CKKS-based performance. LLM inference remains challenging under CKKS, primarily due to substantial memory overhead from CKKS parameter expansion. This necessitates effective model partitioning and efficient ciphertext communication across distributed nodes. Second, leveraging the high concurrency of MKR-generated MetaKernels, we will explore GPU acceleration for CKKS, building on recent advances in GPU-based FHE [Fan et al. 2023; Zhai et al. 2022]. Finally, MKR's unified code template enables high tunability. We plan to develop auto-tuning for various RLWE-based schemes across hardware platforms.

Acknowledgements

We thank all the reviewers for their constructive comments. This research was supported by National Key R&D Program of China (Grant No. 2023YFB4503204).

Data Availability Statement

MKR has been implemented in the open-source ANT-ACE FHE compiler [Li et al. 2025] and is available at https://github.com/ant-research/ace-compiler/tree/metakernel-proof. Our artifact, including source code and raw data for all figures and tables, is available at and a Zenodo link [Yuan et al. 2025].

References

Al Badawi Ahmad, Bates Jack, Bergamaschi Flavio, Cousins David Bruce, Erabelli Saroja, Genise Nicholas, Halevi Shai, Hunt Hamish, Kim Andrey, Lee Yongwoo, Liu Zeyu, Micciancio Daniele, Quah Ian, Polyakov Yuriy, R.V. Saraswathy, Rohloff Kurt, Saylor Jonathan, Suponitsky Dmitriy, Triplett Matthew, Vaikuntanathan Vinod, and Zucca Vincent. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Los Angeles, CA, USA) *(WAHC'22)*. Association for Computing Machinery, New York, NY, USA, 53–63. doi:10.1145/3560827.3563379

Martin R. Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E. Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2019. Homomorphic Encryption Standard. *IACR Cryptol. ePrint Arch.* (2019), 939. https://eprint.iacr.org/2019/939

Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (London, United Kingdom) *(WAHC'19)*. Association for Computing Machinery, New York, NY, USA, 45–56. doi:10.1145/3338469.3358944

Jean-Philippe Bossuat, Rosario Cammarota, Jung Hee Cheon, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Luis Antonio Ruiz Lopez, Yongsoo Song, Donggeon Yhee, and Bahattin Yildiz. 2024. Security Guidelines for Implementing Homomorphic Encryption. Cryptology ePrint Archive, Paper 2024/463. https://eprint.iacr.org/2024/463

Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277

Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. A Full RNS Variant of Approximate Homomorphic Encryption. In *Selected Areas in Cryptography – SAC 2018*, Carlos Cid and Michael J. Jacobson Jr. (Eds.). Springer International Publishing, Cham, 347–368.

Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, Cham, 409–437. doi:10.1007/978-3-319-70694-8_15

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

Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. https://eprint.iacr.org/2018/421

Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 546–561. doi:10.1145/3385412.3386023

Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madan Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *PLDI 2019*. ACM, 142–156. https://www.microsoft.com/en-us/research/publication/chet-an-optimizing-compiler-for-fully-homomorphic-neural-network-inferencing/

Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 617–640.

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

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

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

Craig Gentry. 2009. *A fully homomorphic encryption scheme.* Ph. D. Dissertation. Stanford University.

Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 75–92.

Armin Gerami, Monte Hoover, Pranav S. Dulepet, and Ramani Duraiswami. 2024. FAST: Factorizable Attention for Speeding up Transformers. *CoRR* abs/2402.07901 (2024). arXiv:2402.07901 doi:10.48550/ARXIV.2402.07901

Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 201–210. http://proceedings.mlr.press/v48/gilad-bachrach16.html

Gamze Gürsoy, Eduardo Chielle, Charlotte M. Brannon, Michail Maniatakos, and Mark Gerstein. 2020. Privacy-preserving genotype imputation with fully homomorphic encryption. *bioRxiv* (2020). doi:10.1101/2020.05.29.124412

Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I* (Santa Barbara, CA, USA). Springer-Verlag, Berlin, Heidelberg, 93–120. doi:10.1007/978-3-319-96884-1_4

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV] https://arxiv.org/abs/1704.04861

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

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. doi:10.1145/2647868.2654889

Zhennan Qin Jianhui Li and Dan Lavery. 2024. oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–204.

Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: a low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1651–1668.

Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. In *Proceedings of the 45st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '24)*. ACM. doi:10.1145/3656382

Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022b. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 12403–12422. https://proceedings.mlr.press/v162/lee22e.html

Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison. *IEEE Transactions on Dependable and Secure Computing, 19(6)*, 3711–3727.

Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. 2022a. Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network. *IEEE Access* 10 (2022), 30039–30054. doi:10.1109/ACCESS.2022.3159694

Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. 2025. ANT-ACE: An FHE Compiler Framework for Automating Neural Network Inference. In *2025 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. 2025. ReSBM: Region-based Scale and Minimal-Level Bootstrapping Management for FHE via Min-Cut. In

Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 924–939. doi:10.1145/3669940.3707276

Guiwen Luo, Shihui Fu, and Guang Gong. 2023. Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 2 (2023), 358–380. doi:10.46586/TCHES.V2023.I2.358-380

Meta. 2024. Build the future of AI with Meta Llama 3. https://llama.meta.com/llama3/

Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/3466752.3480070

Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. doi:10.1145/3470496.3527393

SEAL 2020. Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

Halevi Shai and Shoup Victor. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*, Juan A. Garay and Rosario Gennaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 554–571.

Xing Su, Xiangke Liao, and Jingling Xue. 2017. Automatic Generation of Fast BLAS3-GEMM: A Portable Compiler Approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–204.

Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2022. HECO: Automatic Code Optimizations for Efficient Fully Homomorphic Encryption. (2022). https://arxiv.org/abs/2202.01649

Lee Yongwoo, Heo Seonyeong, Cheon Seonyoung, Jeong Shinnung, Kim Changsu, Kim Eunkyung, Lee Dongyoon, and Kim Hanjun. 2022. HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–204. doi:10.1109/CGO53902.2022.9741265

Feng Yu, Guangli Li, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2024. Optimizing Dynamic-Shape Neural Networks on Accelerators via On-the-Fly Micro-Kernel Polymerization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 797–812. doi:10.1145/3620665.3640390

Peng Yuan, Yan Liu, JianXin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Xiaojing Zhang, Qing Zhu, and Jingling Xue. 2025. *MetaKernel Artifact*. doi:10.5281/zenodo.16911192

Y. Zhai, M. Ibrahim, Y. Qiu, F. Boemer, Z. Chen, A. Titov, and A. Lyashevsky. 2022. Accelerating Encrypted Computing on Intel GPUs. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 705–716. doi:10.1109/IPDPS53621.2022.00074

Zhongcheng Zhang, Ying Liu, Yuyang Zhang, Zhenchuan Chen, Jiacheng Zhao, Xiaobing Feng, Huimin Cui, and Jingling Xue. 2025. Qiwu: Exploiting Ciphertext-Level SIMD Parallelism in Homomorphic Encryption Programs. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 523–537. doi:10.1145/3696443.3708917

Received 2025-03-26; accepted 2025-08-12