FHE-CGRA: Enable Efficient Acceleration of Fully Homomorphic Encryption on CGRAs

Miaomiao Jiang
Shandong University
Quan Cheng Laboratory
202316985@mail.sdu.edu.cn

Yilan Zhu
Ant Group
Shandong University
201911906@mail.sdu.edu.cn

Honghui You
Shandong University
Quan Cheng Laboratory
yhh21@mail.sdu.edu.cn

Cheng Tan
Arizona State University
chengtan@asu.edu

Zhaoying Li
National University of Singapore
zhaoying@comp.nus.edu.sg

Jiming Xu
Ant Group
jiming.xjm@antgroup.com

Lei Ju*
Quan Cheng Laboratory
sr-julei@qcl.edu.cn

ABSTRACT

Fully Homomorphic Encryption (FHE) is an attractive privacy-preserving technique that allows computation directly on encrypted data without decryption. However, it incurs significant performance and memory costs due to intensive computations. In this work, we investigate the execution of FHE-enabled machine learning (ML) applications. We show that the runtime hardware reconfigurability of the underlying execution units of homomorphic operations is highly desirable for efficient hardware resource utilization during FHE-ML execution, due to the changing FHE encryption variants across different ML stages (e.g., the multiplicative level of the ciphertext) and corresponding optimal execution unit design. Based on the observation, we propose FHE-CGRA, a coarse-grained reconfigurable architecture (CGRA) acceleration framework with an MLIR-based compiler toolchain for end-to-end homomorphic applications. The experiment shows that FHE-CGRA achieves up-to 8.15× speedup against a conventional CGRA baseline for accelerating the inference of FHE-encrypted convolution neural network (FHE-CNN) models, and up-to 16.48× power efficiency w.r.t. the state-of-the-art FPGA-based FHE-CNN accelerator design.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Security and privacy → Cryptography**.

KEYWORDS

Fully homomorphic encryption, Reconfigurable architectures, Hardware acceleration, CGRA Mapping, Design flow

ACM Reference Format:
Miaomiao Jiang, Yilan Zhu, Honghui You, Cheng Tan, Zhaoying Li, Jiming Xu, and Lei Ju. 2024. FHE-CGRA: Enable Efficient Acceleration of Fully Homomorphic Encryption on CGRAs . In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656536

*Corresponding author.

1 INTRODUCTION

Data privacy has become a critical concern when deploying machine learning on cloud or edge devices. Traditional encryption technology requires decryption of the encrypted sensitive inputs (e.g., biomedical data, financial data, secret conversation/prompt from users) before processing them, which is vulnerable to breaches during computation. Fortunately, fully homomorphic encryption (FHE) allows computing on encrypted data directly without decryption, which secures the offloading of data processing tasks including machine learning inferences. However, one of the major obstacles to the practical usage of FHE is its prohibitively expensive computation overhead (i.e., $10,000\times$ over unencrypted computation [18]).

Recently, various hardware accelerators have been proposed to speedup FHE computations, including GPU, ASIC and FPGA solutions [1, 9, 10, 17, 19, 23]. GPU accelerators typically consume several hundred watts of power and are not well-suited for integer-based computations used in homomorphic operations. ASIC accelerators have a fixed number configuration of homomorphic operators and predetermined FHE encryption parameters, which leads to poor programmability for various FHE applications. Significantly, ASIC work F1 [18] achieves a mere 30% utilization rate for its functional units. While FPGAs offer high flexibility to accommodate various FHE applications, they suffer from low efficiency (e.g., in milliseconds) in terms of bit-level dynamic partial reconfigurability. However, literature work reply on a fixed circuit level design to accelerate end-to-end FHE applications, and may suffer from resource under-utilization at various run-time execution stages.

Specifically, we face two challenges when designing an accelerator for FHE applications where run-time reconfigurability is highly desired. Firstly, accelerators are required for accommodating different settings of FHE parameters (e.g., polynomial degree, modulus, and the level of ciphertext). This leads to huge design space in practice and impacts the computation workload as well as the resource efficiency of the hardware accelerator design. Furthermore, considering the changing FHE variants across different neural network layers and intermediate bootstrapping operations, an immutable hardware design may not lead to optimal performance. Secondly, the flexibility required for FHE applications is reflected in the varying configuration ratio of their underlying homomorphic basic operators, which changes as the computational tasks at runtime. For example, the homomorphic activation layer and the fully connected layer require different basic operators. The former layer requires additional "Automorphism" operations that are not needed in the latter. These differences reflect the need for flexibility in the design of FHE accelerators.

Coarse-grained reconfigurable array (CGRA) becomes a promising architectural choice to meet the design and runtime hardware reconfigurability demands of FHE applications. To the best of the authors' knowledge, this paper proposes the first CGRA acceleration framework for homomorphic operations and end-to-end applications. Specifically, we utilize the characteristics of FHE computations to design configurable CGRA compute engines for generic FHE dialects, and create an MLIR-based compiler toolchain to automate the lowering from a given ML model towards the basic homomorphic operations accelerated on target CGRA platforms. Technical contributions of this work are summarized as follows.

- The paper demonstrates that runtime reconfigurability is an inherent need to build resource-efficient hardware FHE accelerators, particularly for CNN kernels with changing FHE encryption variants across different layers and bootstrapping operations.
- The paper proposes a design of configurable and efficient CGRA primitives with first-order representational support for key FHE dialects, which enables partitioned mapping strategy with high performance and low runtime reconfiguration overhead.
- The paper implements an MLIR-based end-to-end compilation framework that can lower an ML model from various ML frontends (e.g., ONNX, PyTorch, TensorFlow) to the `FHE-dialect`. Automatic compiler optimization techniques (e.g., operation fusion) have been applied to further improve the overall performance of the executable CGRA code.
- Experiment results show that the proposed FHE-CGRA framework achieves up-to 8.15× speedup against a conventional CGRA baseline for accelerating FHE-CNN model inference, and 16.48× power efficiency w.r.t. the state-of-the-art FPGA-based FHE-CNN accelerator design.

2 BACKGROUND AND MOTIVATION

2.1 Fully Homomorphic Encryption

Currently, there are various schemes available for fully homomorphic encryption (FHE), such as BFV [8], TFHE [6] and CKKS [5]. Our work is based on the widely used CKKS [5] scheme which supports floating-point computation. FHE supports homomorphic addition, multiplication, and rotation (moving vector data within ciphertexts). It also includes supplementary operations like relinearization and rescaling. FHE employs bootstrapping to refresh ciphertexts with high noise into low noise for unlimited homomorphic operations. Moreover, all these homomorphic operations are built upon basic operations, such as modular addition, modular multiplication, number theoretic transform (NTT), and automorphisms. Our implementation of these basic operations is compatible with other CKKS-like FHE schemes.

An FHE ciphertext typically consists of polynomials, each of which has N coefficients. Typically, N is a power of 2, ranging from 2^{13} to 2^{17} for security, but this imposes a substantial computational workload. Each coefficient is an integer smaller than the modulus Q, which typically ranges from hundreds to thousands of bits. Using the RNS [3] technique, coefficients can be decomposed into smaller prime modulus q_i, where $Q = \prod_{i=1}^{L} q_i$. Rescaling reduces RNS polynomial of q_i after each multiplication, so the parameter L represents the maximum achievable multiplication depth.

	intt	ntt	ModMult	ModReduce	Automorphism
Conv Layer	5	6	17	6	0
Activation Layer	7	7	15	9	0
FC Layer	6	6	8	8	4

Table 1: Number of basic operation units required in LoLa-MNIST [4]'s first three layers.

2.2 Motivation

FHE applications suffer from poor performance, and demands high computational resource efficiency and flexibility. More specifically, FHE primarily offers homomorphic operations, which are constructed from basic operations such as ntt and modular multiplication. The types and quantities of basic operations vary within different FHE operations, e.g., the Keyswitch operation contains both ntt and intt, whereas homomorphic multiplication only requires modular multiplication. When the number of basic operation units is fixed, the hardware cannot guarantee optimal performance for different FHE operations. Further, FHE-CNN includes homomorphic convolutional layers, fully connected layers, and more. Various layers incorporate distinct FHE operations, consequently resulting in variations in basic operations. Moreover, as the CNN layer computation proceeds and the multiplication depth accumulates, the required Rescaling operation throws away redundant polynomials, resulting in fewer basic operation units being required.

For each basic FHE operation, Table 1 displays the maximum number of instances that can be executed in parallel in the first three layers of LoLa-MNIST [4]. The Conv layer has three times the modular multiplications of ntt, while the FC layer shows a similar quantity. Automorphism units are only used in FC layer, not in the first two. This highlights the need for adaptability in computations, as a fully connected layer-centric configuration is suboptimal for convolutional layers under resource constraints. CGRAs can be quickly reconfigured (hundreds of cycles, i.e., in nanoseconds [21]) to maximize the hardware utilization for each accelerated kernel.

3 FHE-CGRA ARCHITECTURE AND MAPPING

This section briefly describes the conventional CGRA architecture and introduces the proposed domain-specific architecture features for efficient acceleration of FHE-based machine learning models.

3.1 Conventional CGRA

Figure 1(a) shows a typical system integrating a CGRA accelerator. The data (i.e., inputs, outputs, and control signals) transfer between the CGRA and the host is enabled by a direct memory access (DMA) unit. The CPU can invoke the CGRA via the accelerator command interface. The scratchpad memory (SPM) of the CGRA can be accessed via a set of tiles (usually the left-most tiles). Traditionally, CGRA is homogeneous design that contains identical tiles interconnected through a mesh on-chip network. Each tile contains a set of functional units, registers, a crossbar, and a control memory. Given an application kernel, the CGRA compiler maps the kernel onto

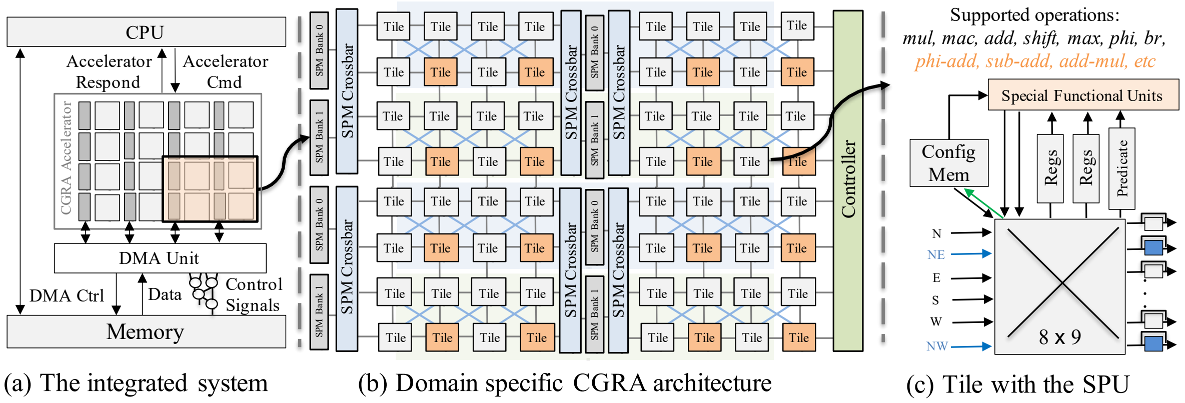


Figure 1: FHE-CGRA system architecture – A heterogeneous CGRA architecture adapted to FHE-based machine learning workload.

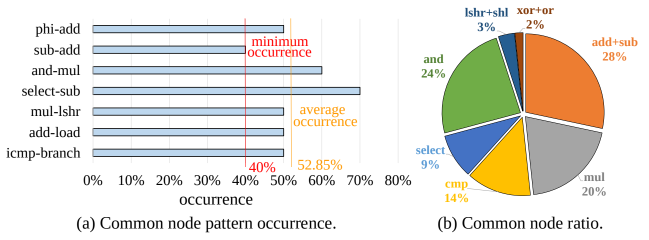


Figure 2: Common node pattern occurrence and node ratio across all basic FHE operations – `phi-add` indicates a phi operation followed by an addition.

the CGRA tiles and generates the control signals of the mapping. Before the execution, the control signals of the target kernel are loaded into the control memory in each tile to guide the dataflow and execution on each tile at every cycle during runtime.

Kernel Mapping – Mapping of CGRA aims to find a mapping with the minimum initiation interval between a Data-Flow Graph (DFG) of a given kernel and a Modulo Routing Resource Graph (MRRG) [15] representing the time-extended resource graph of the CGRA. We implement a heuristic mapper based on [12] that handles mapping strategies specialized for FHE applications.

3.2 Proposed Domain-Specific Architecture Features

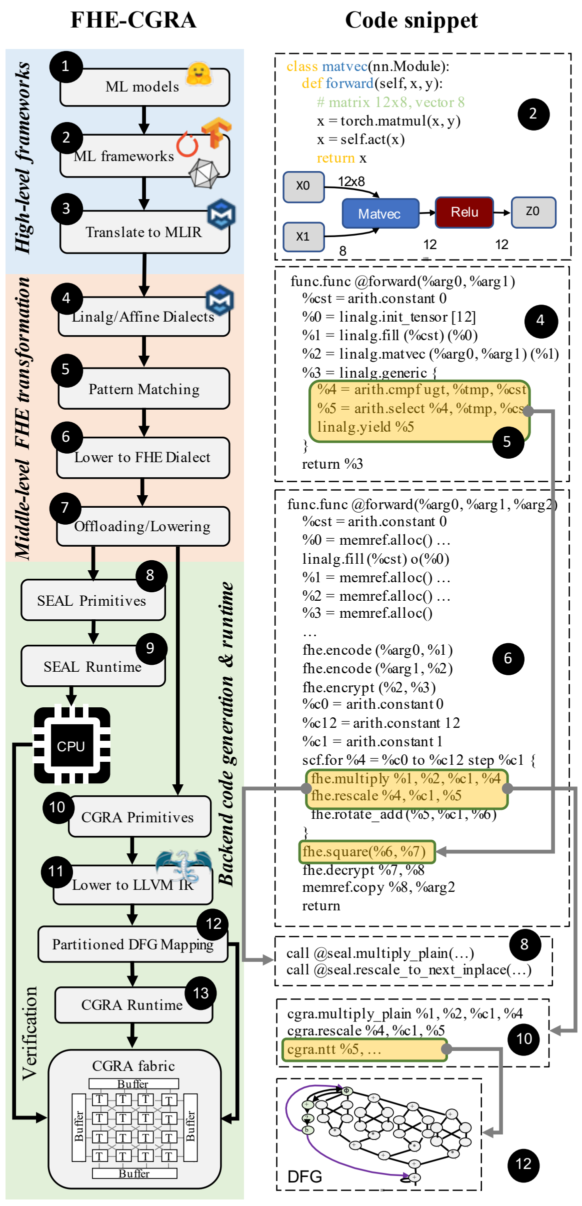
For efficient acceleration of FHE-based ML workloads, we augment the conventional CGRA architecture with additional hardware components (highlighted in Figure 1) without losing its generalization.

Heterogeneity – Instead of the homogeneous CGRA design, we embrace the heterogeneity by customizing a set of tiles based on the FHE operations. By exploiting DFGs produced by FHE operations, we find seven common node patterns (listing in Figure 2(a)) across all the FHE operations and customize the tiles to enable single-cycle execution of them. Figure 2(a) shows the occurrence of the seven node patterns. 64 tiles of FHE-CGRA prototype incorporate the special functional units (SFUs) to support the seven common patterns. During compilation, these patterns are recognized, fused into a single node in DFG, and mapped onto the tiles equipped with the SFU. This results in a reduction of nodes in the DFG, leading to faster processing. To facilitate the mapping of these patterns (i.e., enable the required data to be successfully routed to the customized tiles), we increase the ports of the crossbars on certain tiles (highlighted in blue).

In Figure 2(b), "mul" and "add+sub" nodes constitute approximately half of the overall workload in the FHE while others either account for a low proportion or require fewer computational resources. To cater to the specific needs of FHE applications, we tailor the tile distribution to accommodate the varying ratios of these nodes within FHE-CGRA.

Partitioned Mapping – The basic FHE operations can be represented as DFGs with hundreds of nodes. Existing CGRA mapping strategies [14] suffer from mapping large DFG (e.g., more than 100 nodes) onto big CGRA fabric (e.g., more than 6×6 tiles), which usually leads to sub-optimal acceleration speedup and unaccepted compilation time (e.g., hours) [12, 22]. To facilitate the mapping and maximize the speedup, we limit the search space for the mapping algorithm by partitioning the tiles into small groups (each contains fewer tiles) which reduces the resources required, resulting in different MRRGs. The mapper then can parallel a given kernel in different degrees. A controller is implemented to feed the control signals and each of the partitions can perform on one instance of the target kernel. For certain FHE operations, such as homomorphic addition, parallelization in partitioned CGRA is a natural solution, as there is no recurrence in data. However, for operations that involve significant overlapping memory access, such as ntt, we employ the algorithm optimization [18] to enable partitioning. The evaluation shows such partitioned mapping that partitions FHE-CGRA into groups of 4×4 tiles improves the speedup by 5.88× for the ntt operation compared to the conventional CGRA.

On-Chip Memory – SPM offers input data and result caching for basic operations deployed on CGRA tiles. Our design has a 1.5MB SPM that is split into 16 partitions. Each SPM partition contains 2 banks (two ports for each bank) and each bank can be accessed independently. The SPM partitions are interleaved with the groups of 4×4 tiles, facilitating the data feeding for the partitioned mapping. When performing partitioned mapping of FHE operations, tiles that are close to banks can access data from nearby banks. The SPM incorporates a double buffer to overlap read and write operations efficiently. With a typical 400MHz (1.6GB/s) DMA controller, the data access latency can be hidden under the



```
class matvec(nn.Module):
  def forward(self, x, y):
    # matrix 12x8, vector 8
    x = torch.matmul(x, y)
    x = self.act(x)
    return x
```

```
func.func @forward(%arg0, %arg1)
  %cst = arith.constant 0
  %0 = linalg.init_tensor [12]
  %1 = linalg.fill (%cst) (%0)
  %2 = linalg.matvec (%arg0, %arg1) (%1)
  %3 = linalg.generic {
    %4 = arith.cmpf ugt, %tmp, %cst
    %5 = arith.select %4, %tmp, %cst
    linalg.yield %5
  }
  return %3
```

```
func.func @forward(%arg0, %arg1, %arg2)
  %cst = arith.constant 0
  %0 = memref.alloc() …
  linalg.fill (%cst) o(%0)
  %1 = memref.alloc() …
  %2 = memref.alloc() …
  %3 = memref.alloc()
  …
  fhe.encode (%arg0, %1)
  fhe.encode (%arg1, %2)
  fhe.encrypt (%2, %3)
  %c0 = arith.constant 0
  %c12 = arith.constant 12
  %c1 = arith.constant 1
  scf.for %4 = %c0 to %c12 step %c1 {
    fhe.multiply %1, %2, %c1, %4
    fhe.rescale %4, %c1, %5
    fhe.rotate_add (%5, %c1, %6)
  }
  fhe.square(%6, %7)
  fhe.decrypt %7, %8
  memref.copy %8, %arg2
  return
```

```
call @seal.multiply_plain(…)
call @seal.rescale_to_next_inplace(…)
```

```
cgra.multiply_plain %1, %2, %c1, %4
cgra.rescale %4, %c1, %5
cgra.ntt %5, …
```

Figure 3: FHE-CGRA framework with code snippets.

computation latency using the double buffer. Consider the most memory-intensive operation of FHE, ntt, the computation latency is 0.41 ms and the data access latency is 0.26 ms.

4 FHE-CGRA COMPILATION FRAMEWORK

To automate the acceleration of FHE-based machine learning models on the proposed CGRA, we propose FHE-CGRA compilation framework – an integrated toolchain based on MLIR (Multi-Level Intermediate Representation [13]) as shown in Figure 3. Given a ML model, rather than manually developing the CGRA kernels, the proposed framework eliminates the programmability and removes the gap among the ML models, FHE operations/primitives, and the CGRA targets. Specifically, the proposed framework automatically compiles it into `fhe-dialect` and enables the lowering to the corresponding CGRA primitives that can be offloaded, mapped, and accelerated by the CGRA. The FHE-CGRA framework contains three parts: High-level frameworks, middle-level FHE transformation, and backend code generation and runtime.

FHE dialect and SEAL primitives	CGRA primitives
add	modAdd
multiply	modMult
square	modMult
rescale (retains the scale constant after HE multiply)	ntt intt rescaleCore
relinearize (prevents ciphertext polynomials increase after HE multiply)	ntt intt modMultSub innerProduct
rotate (rotates the slots of the encrypted vector)	ntt intt modMultSub innerProduct automorphism

Table 2: The `fhe-dialect` and the corresponding SEAL and CGRA primitives – *modAdd, modMult, modMultSub* denote the basic operations. *rescaleCore* denotes a set of basic operations during rescale calculation. *innerProduct* performs hadamard products with evaluation keys. *automorphism* is used to permute the ciphertext coefficients.

4.1 High-Level Frameworks

The toolchain can consume a ML model that is described in most popular open-source ML front-end frameworks. One matrix-vector multiplication operation represented in PyTorch is used to demonstrate how the FHE-CGRA framework works. As shown in Figure 3, the PyTorch model is lowered into Linalg dialect of MLIR.

4.2 Middle-Level FHE Transformation

An FHE dialect – We created the `fhe-dialect` and associated passes to automate the lowering from the operation represented in `linalg-dialect` to FHE operations. The `fhe-dialect` is a generic computational dialect with first-order representational support for FHE operations, including homomorphic addition, homomorphic multiplication, relinearization, and rotation, etc. In addition, FHE supports SIMD (Single Instruction, Multiple Data), allowing simultaneous FHE operations on each vector slot (e.g. FHE multiplication). It can significantly reduce the number of FHE operations compared to the original computations. Our compilation process makes use of this feature, performing operations on vectors. It facilitates the automated transformation from the original application to the HE application including CNN, achieving efficient implementation.

Pattern Matching – Given no mature scheme to implement non-linear functions using FHE, most existing FHE strategies (e.g., [11] and [7]) use linear polynomials to approximate the ReLU function. We also adopt the approach same as [11] to use square to replace the activation function. Therefore, we create a pattern matching pass to recognize the activation patterns in `linalg-dialect`, which will be lowered into `fhe.square`. For example, ReLU is represented as `arith.cmp` followed by an `arith.select` (each inside a `linalg.generic` operation), which can be recognized as a `relu` activation by the pattern matching pass.

4.3 Backend Code Generation and Runtime

We provide two sets of primitives (i.e., SEAL primitives and CGRA primitives) that can be targeted by the `fhe-dialect` through lowering. To be specific, SEAL [20] is an easy-to-use open-source (MIT licensed) homomorphic encryption library. SEAL primitives are able to be compiled and linked with the SEAL runtime by our proposed framework to execute on CPU, which can then be used as the golden reference against the CGRA counterparts. On the other hand, the SEAL primitives are not directly suitable for CGRA acceleration as each of them might contain too many (e.g., thousands of) DFG nodes that are not scalable to be mapped as discussed in Section 3. Thus, the FHE-CGRA frameworks lower the `fhe-dialect` into the finer-grained CGRA primitives that consist of the basic FHE operations (e.g., ntt, modular addition, intt), which enables the efficient acceleration thanks to the domain-specific architecture optimizations. Table 2 lists the supported CGRA primitives. For simple FHE dialects like homomorphic add, CGRA primitives stay at the same level of granularity. But for complicated dialects that are unfriendly to CGRA, we separate them into simpler primitives with fewer calculations and data storage. For example, the rescale operation sequentially performs intt, rescaleCore (a set of basic operations in rescale), and ntt to the ciphertext data. We set these basic operations as independent primitives, and after each primitive is computed, CGRA performs reconstruction.

CGRA-Runtime – Once the CGRA primitives are generated, the model can be accelerated on CGRA based on the partitioned mapping (Section 3.2) of each primitive. We create a cycle-accurate CGRA simulator (with a runtime) to connect to FHE-CGRA framework to enable end-to-end automation, simulation, and evaluation.

5 EVALUATION

5.1 Experimental setup

Modeling FHE-CGRA – The performance of each CGRA primitive is obtained from the implemented FHE-CGRA compiler. We also model FHE-CGRA in RTL, synthesize it with Synopsys DC compiler using the 45nm NanGate standard-cell library targeting 800 MHz, and obtain the corresponding power and area statistics for evaluation. We use Cacti6.5 [16] to estimate the power and area of the scratchpad data memory used in FHE-CGRA. The entire work is sized 29.30 mm^2 and consumes up to 5.64 W of power.

Baseline – In our experiments, a 16×16 conventional CGRA is selected as the baseline. We also compare FHE-CGRA with the state-of-the-art work of FPGA [23, 24].

Benchmark – We evaluate FHE operations compared with baseline CGRA to illustrate the potential and advantages of FHE-CGRA. We also evaluate packed bootstrapping and two end-to-end applications. For bootstrapping, we adopt the algorithm from openFHE [2], with N set to 2^{16} and q_i set to 54 bits staying consistent with Poseidon [23]. As for end-to-end applications, we employ the inference of the MNIST and CIFAR10 datasets on the LoLa-MNIST and LoLa-CIFAR10 [4] which are five-layer homomorphic neural networks as FHE-CNN applications. The key distinctions between the two networks lie in layer composition and data volumes.

The FHE parameters of FHE-CNN are set according to [24]. N is 2^{13} for MNIST and 2^{14} for CIFAR10 and the bit-width of q_i is 28 for both. This parameter setting satisfies the security requirements of 128 and 192 bits, respectively.

Latency(s) / Speedup	**FHE-CGRA**	**FxHENN (XCZU15EG)**	**FxHENN (XCZU9EG)**	**Poseidon (U280)**
LoLa-MNIST	0.11	0.19/1.73×	0.24/2.18×	-
LoLa-CIFAR10	27.33	54.1/1.98×	254/9.29×	-
Bootstrapping	0.48	-	-	0.12745
PPW(s^{-1} W^{-1}) / Power Efficiency	**FHE-CGRA**	**FxHENN (XCZU15EG)**	**FxHENN (XCZU9EG)**	**Poseidon (U280)**
Power(W)	5.64	10	10	47.86191
LoLa-MNIST	1.61	0.53/3.04×	0.42/3.86×	-
LoLa-CIFAR10	0.0065	0.0018/3.51×	0.0004/16.48×	-
Bootstrapping	0.37	-	-	0.163934/2.25×

Table 3: Performance and power-efficiency of two FHE-CNN [4] and Bootstrapping on FxHENN, Poseidon and FHE-CGRA – PPW indicates performance-per-watt.

5.2 Performance and Efficiency

To the best of the authors' knowledge, our work is the first CGRA acceleration framework for FHE operations and end-to-end applications. We compare the latency and power efficiency (performance-per-watt) of FHE applications generated by our framework with the state-of-the-art FPGA implementations on various benchmarks [23, 24]. FxHENN [24] leverages low-power boards such as XCZU15EG and XCZU9EG, which have comparable power consumption to our power-efficiency design. In contrast, Poseidon utilizes Xilinx Alveo U280, a high-end accelerator card, consuming 47.9 W of power.

Table 3 shows that our implementation outperforms compared to other works. In LoLa-MNIST benchmark, FHE-CGRA surpasses the XCZU15EG by 1.73× in latency and the XCZU9EG by 2.18×. For LoLa-CIFAR10 benchmark, it achieved 1.98× and 9.29× improvements compared to the XCZU15EG and XCZU9EG, respectively. The average 3.795× improvement demonstrates the superior performance of FHE-CGRA on low-power embedded devices. For Bootstrapping, our performance falls short compared to Poseidon [23], which utilizes the higher power Xilinx U280 FPGA. The performance gap can be attributed to Poseidon's utilization of off-chip memory with a bandwidth of 460 GB/s, which provides a significant advantage. Our work targets the embedded domain acceleration for enhancing power efficiency and employs DMA with a bandwidth of 1.6 GB/s. In this context, our latency achieves a comparable level to that of Poseidon. Note that the DRAM access latency is taken into consideration, which is hidden by double buffering.

Our work exhibits notable advantages in terms of power efficiency. Compared to FxHENN, we achieved a 3.04× to 16.48× improvement, while compared to Poseidon it achieved a 2.25× improvement. This is due to the optimization of the heterogeneity analysis of homomorphic implementation on FHE-CGRA, which increases the reuse of resources on tiles through CGRA's reconfigurability. As a result, it reduces the power and area of FHE-CGRA. Furthermore, our approach incurs minimal reconfigurability costs thanks to the low-cost reconfigurability of CGRA. This highlights the strengths and immense potential of utilizing CGRA for FHE acceleration.

5.3 FHE-CGRA Specifics

Heterogeneity Analysis – With FHE-CGRA, we obtain significant improvement compared with baseline CGRA. Table 4 illustrates the time breakdown of CGRA primitives of LoLa-MNIST and normalized speedup of FHE-CGRA compared with the baseline. In particular, the baseline CGRA is a 16×16 conventional CGRA architecture that is homogeneous and not optimized for FHE. To

	Time cost proportion	FHE-CGRA-	FHE-CGRA
modAdd	0.20%	2.00 ×	18.67 ×
modMult	1.27%	1.09 ×	4.40 ×
rescaleCore	1.57%	1.21 ×	6.52 ×
ntt	42.71%	1.08 ×	5.88 ×
intt	26.30%	1.30 ×	5.50 ×
modMultSub	4.61%	2.00 ×	15.89 ×
innerProduct	2.79%	2.00 ×	16.89 ×
automorphism	14.84%	1.50 ×	24.00 ×
communication	5.71%	-	-
LoLa-MNIST	**100%**	**1.20 ×**	**8.15 ×**

Table 4: CGRA primitives time breakdown of LoLa-MNIST and normalized speedup against the baseline (a 16×16 conventional CGRA without any optimization).

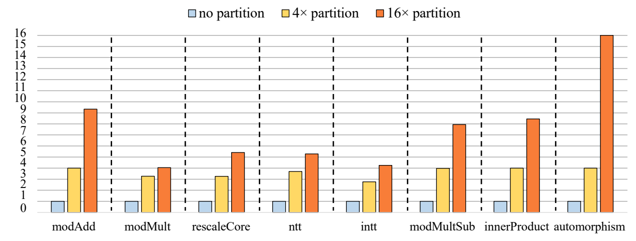


Figure 4: Speedup of the partitioned mapping.

evaluate the performance of specifics, we present two different FHE-CGRA designs: 1) FHE-CGRA-: baseline CGRA with heterogeneity only, and 2) FHE-CGRA: the proposed FHE-CGRA with heterogeneity and partitioned mapping. It is encouraging that FHE-CGRA- 1.2× higher speedup for LoLa-MNIST and an average 1.52× speedup across all CGRA primitives, proving the effectiveness of the heterogeneity optimization technology upon baseline CGRA.

Partition Analysis – We explore the impact of partitioning on CGRA primitives by analyzing the performance of different partition degrees. As shown in Table 4, FHE-CGRA reaches 8.15× speedup for LoLa-MNIST, while FHE-CGRA- only achieves 1.20×. The difference lies in the optimization of partition.

Figure 4 illustrates the speedup of primitives in three scenarios: no partition, 4× partition, and 16× partition. To clarify, the 4× partition involves dividing a 16×16 tile into two 8×8 tiles, and the 16× partition means dividing it into four 4×4 tiles. The increasing partitioning results in optimal utilization of tiles and enhanced performance when the total number of tiles remains constant. The average speedup of 4× and 16× partition reaches 3.62× and 7.59× respectively. These findings confirm the effectiveness of partitioning, leading us to adopt a 16× partition configuration in our design.

6 CONCLUSION

In this paper, we propose FHE-CGRA, a customized CGRA acceleration framework targeting FHE applications. We propose an end-to-end MLIR-based compiler toolchain to automate the lowering from a given ML model towards the basic FHE operations accelerated on the CGRA. The experiment shows that FHE-CGRA achieves 8.15× speedup against a conventional CGRA baseline for accelerating the inference of FHE-CNN models, and 16.48× power efficiency w.r.t. the state-of-the-art FPGA-based FHE-CNN accelerator design.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China grant No.62372272, Department of Science&Technology of Shandong Province grant No.SYS202201, Quan Cheng Laboratory grant No.QCLZD202302, CCF-AFSG RF20220011, Taishan Scholars Program No.tsqn202211281, and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. In *HPCA*. 882–895.

[2] Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *WAHC*. 53–63.

[3] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *SAC*. 423–442.

[4] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low Latency Privacy Preserving Inference. In *ICML*. 812–821.

[5] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*. 409–437.

[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* (2020).

[7] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. 2018. Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference. *CoRR* (2018).

[8] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* (2012), 144.

[9] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. TensorFHE: Achieving Practical Computation on Encrypted Data Using GPGPU. In *HPCA*. 922–934.

[10] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. 2023. BASALISC: Programmable Hardware Accelerator for BGV Fully Homomorphic Encryption. *CHES* (2023).

[11] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*. 201–210.

[12] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A cgra with reconfigurable single-cycle multi-hop interconnect. In *DAC*. 1–6.

[13] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. 2–14.

[14] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. Lisa: Graph neural network based portable mapping on spatial accelerators. In *HPCA*.

[15] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE proc., Comput. digit. tech* (2003).

[16] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. Cacti 6.0: A tool to model large caches. HP laboratories, Tech. Rep.

[17] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *ASPLOS*. 1295–1309.

[18] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO*.

[19] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. 2022. CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA*. 173–187.

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

[21] Cheng Tan, Nicolas Bohm Agostini, Tong Geng, Chenhao Xie, Jiajia Li, Ang Li, Kevin J Barker, and Antonino Tumeo. 2022. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *HPCA*. 304–316.

[22] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2020. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *ICCD*. 381–388.

[23] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *HPCA*. 870–881.

[24] Yilan Zhu, Xinyao Wang, Lei Ju, and Shanqing Guo. 2023. FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference. In *HPCA*. 896–907.