Alchemist: A Unified Accelerator Architecture for Cross-Scheme Fully Homomorphic Encryption

Jianan Mu[1,2,3], Husheng Han[1,2,8], Shangyi Shi[1,2,8], Jing Ye[1,2,3], Zizhen Liu[1,3], Shengwen Liang[1,2,7], Meng Li[4], Mingzhe Zhang[5], Song Bian[6], Xing Hu[1,2,7], Huawei Li[1,2,3], Xiaowei Li[1,2,3]

[1]SKLP, ICT, CAS, [2]University of Chinese Academy of Sciences, [3]CASTEST, [4]Peking University, [5]IIE, CAS, [6]Beihang University, [7]ZGC LAB, [8]Cambricon Technologies

ABSTRACT

The use of cross-scheme fully homomorphic encryption (FHE) in privacy-preserving applications present to be a new challenge to hardware accelerator design. Existing accelerator architectures with customized polynomial-level operator abstraction fail to efficiently handle hybrid FHE schemes due to the mismatch between computational demands and available hardware resources under various parameter settings. In this work, we propose a new accelerator architecture that consists of a novel finer-grained low-level operator, i.e., Meta-OP, that not only mathematically supports a diverse range of polynomial operations, but is also hardware-friendly for accelerator design without complex topological logic. We then design a new slot-based data management scheme to efficiently handle the distinct memory access patterns over the Meta-OP. With a slot-based data management approach, Alchemist can accelerate both arithmetic and logic FHE workloads with high hardware utilization rates. In the experiment, we show that Alchemist is up to 24,829× faster than CPU. For arithmetic FHE, compared with the SOTA ASIC accelerators, Alchemist achieves a 29.4× performance per area improvement on average. For logic FHE, compared with the SOTA ASIC accelerators, Alchemist achieves a 7.0× overall speed up on average.

1 INTRODUCTION

Fully homomorphic encryption (FHE) enables computations to be carried directly on encrypted data, serving as a crucial solution for privacy-preserving computations [1]. Existing FHE schemes can be roughly classified into two categories: arithmetic FHE (such as BFV [2], CKKS [3]) and logic FHE (such as TFHE [4]). Arithmetic FHE schemes enable efficient SIMD-style homomorphic arithmetic operations (e.g., addition and multiplication) by packing multiple plaintexts into a single ciphertext, but are not good at handling non-polynomial functions, like comparison, min/max, and division. In contrast, logic FHE schemes support arbitrary functions represented as boolean circuits by programmable bootstrapping but become extremely inefficient when dealing with large-scale multiplication and addition operations. How to utilize the advantage of both arithmetic and logic schemes is important to enhance the performance and security of secure computations [5]. The recent trending development of algorithm tools that allow for ciphertext switching between two schemes takes advantage of hybrid schemes and improves privacy computation performance [6].

Although from a mathematical perspective, both arithmetic and logic FHE schemes are built upon similar polynomial operators such as number theoretic transform (NTT), RNS Basis conversion (Bconv), decomposed polynomial multiplication (DecompPolyMult, involves the accumulation of decomposed ciphertext multiplied by evaluation key (evk) polynomials), it is still challenging to support cross-schemes FHE acceleration with high hardware efficiency on an architecture level, mainly due to the following reasons: 1) The ciphertext operations and parameter space of the two schemes differs significantly (ciphertext size differences between these two schemes may span over 1000× [3, 4]), which leads to different computing requirements under diverse architecture behavior. 2) The NTT, Bconv and DecompPolyMult operators and their proportions in FHE computations vary significantly. Taking CKKS and TFHE with different parameter sets as examples in Figure 1, the proportion of NTT, Bconv, and DecompPolyMult in different schemes exhibit distinct distributions. Even within CKKS, there are notable variations in the proportions of the three operators for different multiplication depths of the ciphertext.

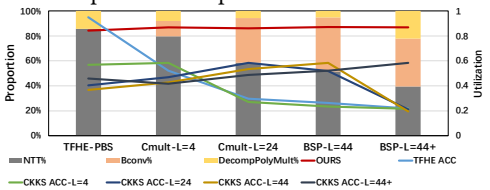

Figure 1: Operator ratio in the algorithm and the overall hardware utilization of different accelerators. TFHE-PBS is TFHE programmable bootstrapping. Cmult-L=n and BSP-L=n are CKKS ciphertext multiplication and bootstrapping with level n. BSP-L=n+ refers to Bootstrapping using Modup hoisting.

Subsequently, the key challenge faced by existing ASIC accelerators [7–11] still is maintaining high hardware utilization rates over various sets of cross-scheme FHE workloads. As shown in Figure 1, since different FHE schemes have divergent high-level operator invocation behaviors and distinct data access patterns, none of the existing ASIC designs can simultaneously achieves high hardware utilization rates overall workload benchmarks. In particular, the varying proportions of NTT, Bconv, and DecompPolyMult of TFHE and CKKS frequently result in a mismatch between computational workload and available hardware resources in existing accelerator designs, causing low hardware utilization rates and inadequate acceleration efficacy.

To this end, we propose Alchemist, a unified accelerator architecture for cross-scheme FHE workloads. To design a unified architecture that meets various FHE workload demands, we first extract a

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657331

key low-level operator, Meta-OP, that can mathematically represent all high-level polynomial operators while being hardware-friendly for architectural integration. Next, by proposing a new data management procedure, we are able to eliminate runtime data exchanges between the Meta-OPs, enabling highly parallelized hardware processing units to be designed. Finally, we integrate such process units into the unified Alchemist architecture alongside the specialized data management procedure. Consequently, as depicted in Figure 1, Alchemist is able to keep high utilization rate across workloads that adopts different FHE schemes. In fact, our accelerator even achieves a higher utilization rate than the operator-based spatial-multiplexing accelerator designed specifically for the FHE workload. The main reason is that such workloads induce a large number of data dependencies, which lead to underutilization of computational resources even in specialized accelerator designs [8, 10, 11]. The main contributions are summarized as follows:

- We identify Meta-OP as the fundamental building block for efficient FHE accelerator design. Meta-OP is finer-grained operator abstraction that is both mathematically complete and hardware friendly.
- We propose the Alchemist architecture design to accelerate Meta-OP and efficiently handle the distinct memory access patterns of Meta-OP. With a slot-based data partitioning strategy, Alchemist supports both arithmetic and logic FHE schemes with high hardware utilization rates.
- Our comprehensive evaluation of Alchemist demonstrates that our design achieves an average performance improvement of more than 7.0× compared to existing designs for arithmetic and logical FHE schemes.

2 BACKGROUND

2.1 Notation

The symbols used in this work and the corresponding definition are shown in Table 1.

Table 1: Symbols and notions in this paper.

Symbol	Definition	Scheme
Q	(Prime) moduli product $\prod_{i=0}^{L-1} q_i$	arithmetic
P	Special (prime) moduli product $\prod_{i=0}^{K-1} p_i$	arithmetic
L	Maximum (multiplicative) level	arithmetic
K	Number of special moduli	arithmetic
N	Degree of a polynomial	both
dnum, l_b	Decomposition number in CKKS, TFHE	both
q	(Prime) moduli	both
M, A, R	Mult, Add, Reduction operators	both

2.2 Operators in Arithmetic and Logic FHE

Both arithmetic FHE and logic FHE rely on underlying polynomial operations. In arithmetic FHE, taking CKKS as an example, the core operations include NTT, Bconv, Modup/down utilizing Bconv, as well as modular multiplication and addition. The polynomial lengths typically range between 2^{10} and 2^{16}, depending on the parameter settings for specific applications. In contrast, logic FHE, exemplified by TFHE, primarily involves NTT, modular multiplication, and modular addition operations. The polynomial lengths are typically set at $2^{10}, 2^{11}, 2^{14}$.

Residue Number System (RNS): In arithmetic FHE, RNS decomposition is used to split ciphertext polynomials with larger modulus Q in 100s or even 1000s of bits into parallel channels with smaller moduli q_i. As a result, Modup/down operations in arithmetic FHE involve plenty of RNS Bconv. Equation (1) describes the RNS Bconv, generating a new channel with the modulus of p_j from modulus Q.

$$Bconv([x]_Q, p_j) : [x]_{p_j} = (\sum_{i=0}^{L-1} [[x]_{q_i} \cdot \hat{q}_i^{-1}]_{q_i} \cdot \hat{q}_i) \bmod p_j \quad (1)$$

Utilizing Bconv in equation (1), equations (2) and (3) depict the process of Modup from modulus Q to the modulus $Q \cdot P$, and Moddown from the modulus $Q \cdot P$ space to Q, respectively.

$$Modup([x]_Q, Q \cdot P) : [x]_{p_j} = Bconv([x]_Q, p_j), j \in [0, K). \quad (2)$$

$$\begin{aligned} &Moddown([x]_{Q \cdot P}, Q) : \\ &[x]_{q_i} = ([x]_{q_i} - Bconv([x]_P, q_i)) \cdot P^{-1} \bmod q_i, i \in [0, L). \end{aligned} \quad (3)$$

Number Theoretic Transform (NTT): NTT is utilized to reduce the complexity of polynomial multiplication from $O(n^2)$ to $O(n \cdot \log n)$. The computation in NTT involves butterfly operations of $a_0 = (a_0 + a_1 \cdot \omega) \bmod q$ and $a_1 = (a_0 - a_1 \cdot \omega) \bmod q$ where a_0 and a_1 are polynomial coefficients and ω is a pre-computed input.

Decomposed polynomial multiplication (DecompPolyMult): DecompPolyMult denotes the procedure of accumulating the decomposed ciphertext multiplied by evaluation key (evk) polynomials. More details can be found in [3, 4].

Modular reduction: FHE ciphertexts are built upon polynomial rings, where the results of addition and multiplication need to be reduced into $[0, q)$. The modular reduction of addition is performed as $a + b = (a + b \geq q)?(a + b - q) : (a + b)$. For multiplications, fast modular reduction like Barrett modular reduction [12] is employed, incorporating a dataflow where 2 multiplications are included.

2.3 Related Works

Accelerators for arithmetic FHE schemes: Recently, FPGA accelerators have been proposed [13–15] and provide a speedup of several hundreds of times compared to CPU. ASIC accelerator designs first target a smaller parameter set [7] and later designs scaled to larger parameter sets [8–10], achieving thousands of performance improvements compared to CPU. More recently, [11] searched for a better RNS word size, 36 bit, achieving the fastest computation speed (we adopt this finding in our design). Moreover, a hardware-agnostic scheduling algorithm was proposed in [16].

Accelerators for logic FHE schemes: To speed up logic FHE computation, AISC architectures are proposed in [17, 18]. They achieve about 1k× throughput improvements over CPU.

As analyzed before, existing accelerators utilize a modularized design approach and have limitations in supporting FHE schemes with varying ratios of different operations.

Although this work is based on the existing memory production processes, some recent works use processing-in-memory (PIM) to improve cryptographic computation speeds. For example, Zhang et al. proposed the first MRAM-based PIM accelerator for LPN cryptography [19]. The PIM-LPN architecture can carry out the entire computations of LPN in memory with zero bit error rate, and achieve about 20× to 200× performance improvement than existing CPU and FPGA implementations.

3 ALCHEMIST OVERVIEW

The overview of our solution is depicted in Figure 2. To efficiently support computations of both arithmetic and logic FHE schemes,

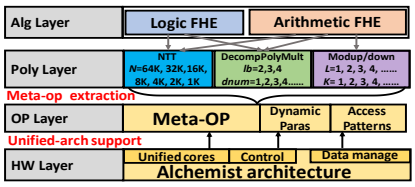

Figure 2: The overview of Alchemist.

we incorporate an OP layer between the computation and hardware layers, rather than directly modularizing the hardware design for different polynomial computations.

At the OP layer, we extract a Meta-OP with dynamic parameters and three customized access patterns for diverse polynomial computations and the huge parameter space of the two FHE schemes: NTT with polynomial length $N \in [2^{10}, 2^{16}]$, DecompPolyMult with decomposition number $lb = 2, 3, 4$ in TFHE and $dnum = 1, 2, 3 \cdots$ in CKKS, and Modup/down with number of input channels $L = 1, 2, 3, \cdots$ and number of output channels $K = 1, 2, 3, \cdots$. By using the Meta-OP in conjunction with three data access patterns, we can support computations for all FHE schemes. Next, at the hardware layer, we design unified computation cores together with dataflow control to support Meta-OP with dynamic parameters. We employ a unified and efficient data management that supports all three memory access patterns.

4 META-OP EXTRACTION

4.1 Challenge and Meta-OP Extraction

The NTT, Modup/down, and DecompPolyMult share the same operator space of addition, multiplication, and reduction. However, they have distinct invocation orders, and their parameter settings differ significantly. In Figure 3, we represent these operations as a function F that includes operations multiplication M, addition A, reduction R, order O, and parameter P. We adjust the order O while ensuring the correctness, and then extract a Meta-OP, denoted as $(M_jA_j)_nR_j$. In $(M_jA_j)_nR_j$, j multiplications and j additions are repeated for n times, the accumulated results are then reduced, as shown in Figure 3. Within $(M_jA_j)_nR_j$, n and j are determined by the parameter P of diverse operators. n is a dynamically adjustable parameter during runtime according to the operation flow, while j is a static parameter, indicating the degree of parallelism of a single Meta-OP. Through an exploration of the design space, we have discovered that setting $j = 8$ is efficient.

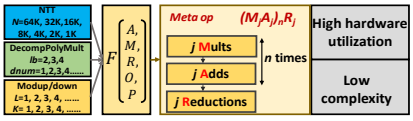

Figure 3: Extraction of the Meta-OP.

This Meta-OP can support all operations across the two FHE schemes and leads to low multiplication complexity and high hardware utilization. Next, we demonstrate the utilization of Meta-OP to perform different polynomial computations

4.2 Polynomial Operations Using Meta-OP

DecompPolyMult: For the DecompPolyMult, Figure 4(a) illustrates the distinct processes of the original operation and the equivalent transformation using $(M_jA_j)_nR_j$ where n is the decomposition number of the ciphertext. Using Meta-OP $(M_jA_j)_nR_j$, the reduction in DecompPolyMult is postponed to the result of the accumulated multiplication product as shown in Figure 4(a).

Table 2 lists the equation and the multiplication overhead of the original operation and the equivalent transformation for decomposition number bing $dnum$. As shown in Table 2, with using the Meta-OP, the number of multiplication is reduced by up to 3× due to the decrease of the number of reduction. Such an adjustment would result in doubling the size of the modular addition, but this is minuscule compared to the large savings in multiplication overhead.

Table 2: Transformation of DecompPolyMult.

Diagram	Equation	#Mults
Origin	$\sum_{i=1}^{dnum}(\text{Reduce}_q(a_i \cdot b_i))$	$3 \cdot dnum \cdot N$
$(M_jA_j)_{dnum}R_j$	$\text{Reduce}_q(\sum_{i=1}^{dnum}(a_i \cdot b_i))$	$(dnum + 2) \cdot N$

In $(M_jA_j)_nR_j$ for DecompPolyMult, the accumulative products come from different "dnum groups". Meanwhile, using Meta-OP $(M_jA_j)_{dnum}R_j$ for DecompPolyMult, a high utilization rate can be achieved as long as j can divide N (N is a power of 2).

Modup/down: Similarly, we apply the Meta-OP to Modup/down. In Figure 4(b), we demonstrate the transformation using modup with $L = 2$ and $K = 2$ as an example. As shown in Figure 4(b), modup can be divided into two steps: in step 1, the computation of aq is performed independently in each channel, and in step 2, for each target channel, all L aqs are multiplicated by the constants, reduced and aggregated. Using our Meta-OP $(M_jA_j)_LR_j$, the number of reductions in the aggregation of the modular product is decreased. The general equation of the transformation of Modup is listed in Table 3. During the Modup/down computing, a high utilization rate can be achieved as long as j can divide N.

Table 3: Transformation of Modup.

Diagram	Equation	#Mults
Origin	$a\hat{q}_i^{-1} = \text{Reduce}_{q_i}(a \cdot \hat{q}_i^{-1}), i \in [0, L)$ $\sum_{i=0}^{L-1}\text{Reduce}_{p_j}(a\hat{q}_i^{-1} \cdot \hat{q}_i), j \in [0, K)$	$(3K \cdot L + 3L) \cdot N$
$(M_jA_j)_LR_j$	$a\hat{q}_i^{-1} = \text{Reduce}_{q_i}(a \cdot \hat{q}_i^{-1}), i \in [0, L)$ $\text{Reduce}_{p_j}(\sum_{i=0}^{L-1}(a\hat{q}_i^{-1} \cdot \hat{q}_i)), j \in [0, K)$	$(K \cdot L + 3L + 2M) \cdot N$

NTT: For NTT operation, we perform the radix-r butterfly operation using $(M_jA_j)_3R_j$ Meta-OP. We illustrate the process in Figure 4(c). As shown in Figure 4(c), all 8 results of the the radix-8 butterfly are composed of three parts: products of a_7 and a_6 with twiddle factor (in red box), products of a_5 and a_4 with twiddle factor (in yellow box), and products of a_3 to a_0 with twiddle factor in blue box). We compute each multiplication in the red, yellow and blue boxes and added them to get each result using a Meta-OP$(M_8A_8)_3R_8$. This computation requires 3×8 = 24 multiplications and 8 reductions, totaling 40 multiplications. Compared to the original NTT that requires 12×3 = 36 multiplications, only causes a 10% multiplication increase. It is worth highlighting that directly unfolding the iterative NTT would lead to a several times multiplication penalty. However, by leveraging our Meta-OP, which reduces the number of reductions, we have successfully minimized the penalty of multiplication overhead to only 10%. In addition, to support all possible polynomial lengths N, we also use the Meta-OP to perform the radix-4 butterfly in a similar way.

Regarding the parameter j, using 16, 32 or other values greater than 8 that divide N would result in low utilization for NTT computation. Therefore, we fix j to 8, ensuring high utilization across all operations in FHE. In addition, we summarize the three kinds

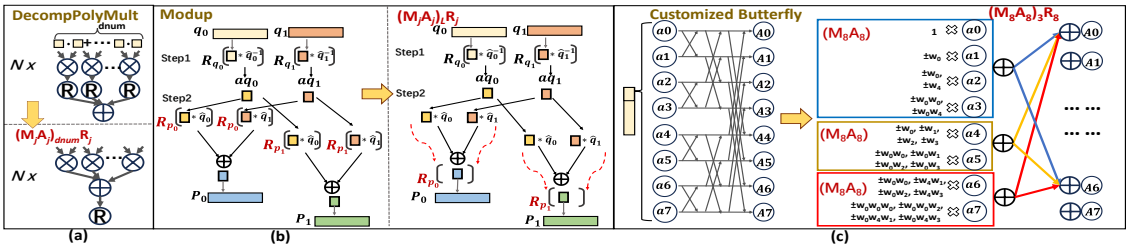

Figure 4: Computations of different operations using $(M_jA_j)_nR_j$: (a) DecompPolyMult. (b) Modup. (c) NTT.

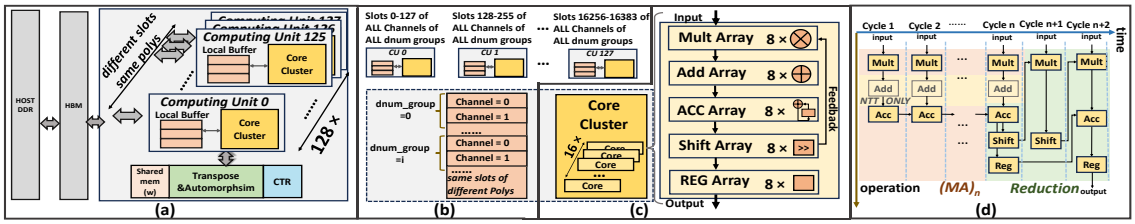

Figure 5: (a) Overall architecture. (b) Data management. (c) Core design. (d) Dataflow control for $(M_8A_8)_nR_8$.

of data access patterns required by our Meta-OP for NTT, Modup/-down, and DecompPolyMult in Table 4.

The main significance of using the Meta-OP to support FHE operations lies in the elimination of the need for dedicated computing modules. Meanwhile, as for the multiplication, our further experiments indicate that using this Meta-OP does not increase but reduces the overall multiplication overhead. It is because although there is a slight multiplication increase in NTT, the number of multiplications in Modup/down and DecompPolyMult is significantly decreased.

Table 4: Data access pattern of the three operations.

	Access pattern		
Computation	Slots	Channel	Dnum_group
(I)NTT	✓	–	–
DecompPolyMult	–	–	✓
Modup/down	–	✓	–

5 UNIFIED HARDWARE DESIGN

5.1 Challenge and Overall Architecture

The primary challenges in hardware design stem from the conflicts between diverse parameter n and three access pattern of $(M_8A_8)_nR_8$ for diverse applications, and the efficiency of the hardware design.

To support the Meta-OP for different applications with a high hardware utilization, we designed a unified core composed of multiplication and addition arrays and decompose the reduction operation into multiplications and additions. To accommodate the three different memory access patterns required for Meta-OP, we employ efficient slot-based data management. We also adopt an on-chip task scheduling approach that optimizes on-chip caching.

The overall architecture is shown in Figure 5(a). It consists of 128 parallel computing units, a shared memory of 2MB, a transpose buffer, and a control unit. The 128 parallel computing units are independent and there is no data exchange between each computing unit. Within each computing unit, there is a local scratchpad and a core cluster. Each local scratchpad is sized 512KB, and the total size of the on-chip scratchpad is 64+2 MB. Each core cluster consists of 16 parallel cores, and each core performs a Meta-OP.

5.2 Unified Core Design

The unified core, as shown in Figure 5(c), consists of a multiplication array, an addition array, an accumulation array, and a register array. Each array contains eight corresponding components.

We do not instantiate modular reduction components, instead, we reuse the mult array and the accumulation array by data flow control. Figure 5(d) illustrates the spatiotemporal data flow during the execution of $(M_8A_8)_nR_8$. The entire computation is temporally divided into two parts: $(M_8A_8)_n$ (the pink part) and reduction (the green part). The $(M_8A_8)_n$ operation requires n cycles, and in each cycle, the multiplication produces 8 results and then accumulates with the previous results. Specifically, for $(M_8A_8)_3$ operation in NTT, results of the 8 multiplications are first passed through an addition array, which recombines them into 8 intermediate results and then accumulated with the previous results. The reduction operation is achieved by reusing the multiplication array as shown in the green part in Figure 5(d), which takes 2 cycles. In total, it takes $n+2$ cycles for the unified core for each Meta-OP $(M_8A_8)_nR_8$. During the whole process, the utilization of the core remains high.

5.3 Data Management

To efficiently accommodate the diverse memory access requirements of NTT, Modup/down, and DecompPolyMult, we use a slot-based data partition and employ a 4-step NTT to maximize data locality. This data organization approach eliminates the need for data exchange between computing units.

Slot-based data partioning: We distribute all polynomials across different computing units by slot. Each unit's local SRAM stores the same slots for all channels of all dnum groups. Figure 5(b) illustrates the data allocation in the local SRAMs, taking the example of $N = 16384$. For each channel of every dnum group, polynomial slots $0-127$ are stored in local SRAM 0, polynomial slots $128-255$ are stored in local SRAM 1, and so on. This arrangement ensures that every channel of every dnum group is stored in each local SRAM, so the data required for DecompPolyMult and Modup/down can be accessed within the private local SRAM.

Maximizing data locality: The classical NTT algorithm is fully connected, which contradicts our slot-based data partition. Therefore, we adopted the 4-step NTT algorithm to maximize data

locality during NTT computations. For instance, in the case of $N = 16384$, the 128×128 slots of each polynomial are evenly partitioned into 128 units, and each computing unit stores 128 slots of each polynomial. Using the 4-step NTT algorithm, the 16384-point NTT is decomposed into 128 sub-NTTs with each sized 128 point. Therefore, each computing unit runs 128-point NTT and all the necessary data are located in the private local SRAM.

By employing slot-based data partitioning and the 4-step NTT algorithm, each core cluster only accesses its own private local SRAM. This data management enables support for the three memory access patterns of Meta-OP, without the need for data exchange between computing units. Consequently, our multi-core accelerator architecture achieves high efficiency.

5.4 Scheduling and Design Space Exploration

Compared to modularized design, our unified architecture enables the decoupling of scheduling from the underlying hardware, eliminating the concern about the availability of individual modules and the overall utilization rate. Based on this, we implemented a time-sharing scheduling strategy similar to [8, 14] to optimize on-chip storage overhead for cross-scheme FHE computing. Through experiments on the benchmark with parameter sets as the same as the latest design [11], in our design, a 64 + 2 MB on-chip SRAM size, effectively eliminates memory access bottlenecks in FHE tasks.

We also conducted a design space exploration to ensure high utilization in the cross-scheme FHE, accommodating a wide range of polynomial lengths N, spanning two orders of magnitude. According to the latest findings in [11], we select a word size of 36 bits for the arithmetic FHE. It also has good compatibility with the TFHE scheme. Ultimately, the current architecture utilizing 128 core clusters was determined.

6 EXPERIMENTAL EVALUATION

6.1 Implementation Results

Table 5: Area breakdown of Alchemist.

Components	Area(mm^2)
1× Core Cluster (16× CORE)	16 × 0.043
1× Local SRAM	1 × 0.427
1× Computing Unit (Core Cluster + Local SRAM)	1 × 1.118
128× Computing Unit	143.104
Register file for transpose	6.380
Shared memory	1.801
Memory interface (2×HBM2 PHYs)	29.801
Total	181.086

We implemented Alchemist in RTL and synthesized it in a commercial 14nm process technology using Design Compiler. We used CACTI to model SRAM components. Two HBM2 stacks are utilized, with a total bandwidth of 1TB/s. The design runs at 1GHz. In total, Alchemist is sized 181mm^2 and consumes 77.9 watts on average. The area breakdown is listed in Table 5.

We also list the resource usage of the four latest accelerators specified for arithmetic (AC) or logic (LC) schemes in Table 6. In Table 6, only our accelerator provides efficient support for both FHE schemes. Moreover, compared with the latest accelerator specified for arithmetic FHE, our SRAM consumption is reduced by more than 60% and the overall area is reduced by more than 50%.

6.2 Evaluation of Benchmarks

To evaluate the performance, we implement a cycle-accurate simulator for Alchemist and test Alchemist with various benchmarks,

Table 6: Rescouces usage in FHE accelerators.

	Matcha [17]	Stritx [18]	CraterLake [10]	SHARP [11]	Alchemist
(AC, LC)	(–,✓)	(–,✓)	(✓,–)	(✓,–)	(✓,✓)
Off-chip mem BW	640 GB/S	300 GB/S	2.4 TB/S	1 TB/S	1 TB/S
On-chip mem Cap	4 MB	26 MB	256 MB	180 MB	66 MB
On-chip mem BW	/	/	84 TB/S	72 TB/S	66 TB/S
Core Freq	2 GHz	1.2 GHz	1 GHz	1 GHz	1 GHz
Area	36.96	141.37		178.8	
(14nm-scaled)	(33.6)	(56.4)	472.3	(379)	181.1

including basic FHE operators, different CKKS applications, and TFHE programmable bootstrapping.

6.2.1 Basic FHE operators. We present in Table 7 the throughput of Alchemist for basic operators with $N = 65536, L = 44$ and $dnum = 4$, and compare it with CPU(Intel Xeon Gold 6234@3.3 GHz with a single thread), GPU [20], and FPGA accelerator [15]. The comparison shows that Alchemist accelerates Pmult and Hadd operations over $20k\times$ compared to the CPU, and speeds up the Keyswitch, Cmult, and Rotation by over $18k\times$.

Table 7: Throughput comparison for basic operators.

	CPU	GPU	Poseidon	Alchemist	Speed up
Pmult	38.14	7,407	14,647	946,970	24,829×
Hadd	35.56	4,807	13,310	710,227	19,973×
Keyswitch	0.4	/	312	7,246	18,115×
Cmult	0.38	57	273	7,143	18,785×
Rotation	0.39	61	302	7,179	18,377×

6.2.2 Benchmark Evaluation. **CKKS Applications:** For shallow CKKS applications, we conducted experiments on LoLa-MNIST [21] with encrypted and unencrypted weights following the latest accelerators [7, 10]. As shown in Figure 6, the results demonstrate that Alchemist achieves over 3× speedup compared to F1. The inference performance with encrypted weights consumes 0.11 ms.

In deep CKKS applications, we test fully-packed bootstrapping and 1024-batched HELR using the same benchmark and parameter settings as the latest design [11]. The comparison results are shown in Figure 6. Alchemist outperforms prior accelerators in both benchmarks. Compared to prior accelerators, Alchemist performs 18.4× (vs. BTS), 6.1× (vs. ARK), 3.7× (vs. CLAKE+), 2.0× (vs. SHARP) faster on average across the two applications. Alchemist also shows significant improvements in the performance per chip area of about 29.4× on average compared to prior accelerators, specifically 76.1× (vs. BTS), 28.4× (vs. ARK), 9.4× (vs. CLAKE+), and 3.79×(vs. SHARP).

TFHE programmable bootstrapping: We evaluated the throughput of programmable bootstrapping in TFHE with two different sets of parameters as the same as [18]. The results indicate that we achieved approximately a 1600× speedup compared to Concrete (CPU) [22] and a 105× speedup compared to NuFHE (GPU) [23]. Compared with the latest TFHE ASIC accelerators [17, 18], we achieved comparable performance per chip area and a 7.0× overall speed up on average.

The experimental results demonstrate that our accelerator efficiently supports cross-scheme FHE computing. Alchemist demonstrates superior improvement in performance and performance per chip area compared to prior accelerators.

Complexity and hardware utilization analysis: Our performance optimization mainly arises from two factors: 1) the reduction in computation overhead, and 2) the increased utilization of hardware resources.

Firstly, we evaluate the number of multiplications in diverse operators in both FHE schemes. The results are shown in Figure 7(a)

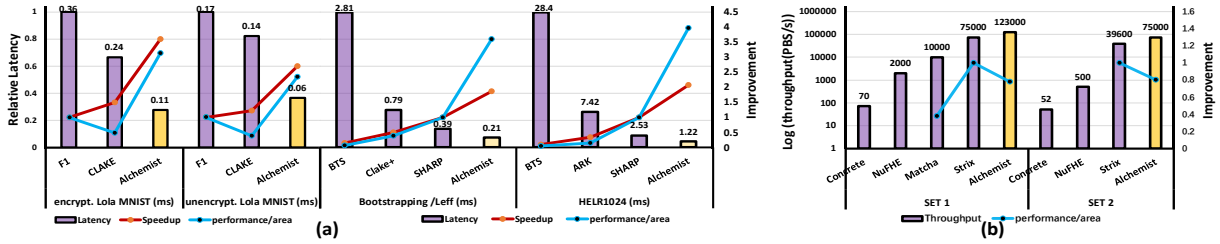

Figure 6: Performance of Alchemist in (a) CKKS applications and (b) TFHE programmable bootstrapping.

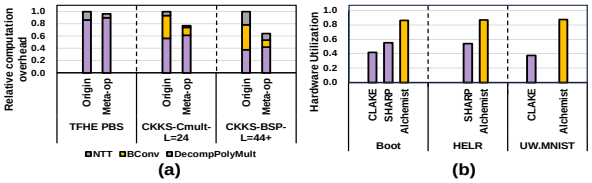

Figure 7: (a) Computation overhead of Cmult and bootstrapping w/ and w/o $(M_j A_j)_n R_j$ and (b) comparisons of utilization rates.

and it demonstrates that due to the benefits derived from Bconv and DecompPolyMult, the overall computation across all applications has been reduced. The multiplication overhead is decreased by 3.4%, 23.3%, and 37.1% for TFHE programmable bootstrapping, CKKS Cmult with $L = 24$ and bootstrapping with $L = 44$ using Modup hoisting.

Secondly, we also evaluate the utilization rates of computational resources in our accelerator and compare it to SOTA designs [10, 11] in Figure 7(b). In [11], during the execution of bootstrapping (HELR-1024), the utilization rates of NTTU, BconvU, and Element-wise Engine are 0.7 (0.68), 0.26 (0.24), and 0.64 (0.53), respectively, with an overall utilization rate of about 0.55 (0.52). In contrast, our design is able to activate the entire on-chip computational resources, where the utilization rate of our design is roughly 0.85, 0.89, and 0.87 on NTT, Bconv and DecompPolyMult tasks, respectively. With an overall utilization rate of about 0.86, we are able to achieve an improvement of utilization rates by approximately 1.57× (1.66×) over SHARP [11]. In addition, our design also enjoys the computation overhead reduction obtained from the unified lazy reduction strategy as described in Table 2, 3 and Figure 7(a). Consequently, in both the bootstrapping (i.e., boot) and HELR applications, we achieve overall performance improvements of 1.85× and 2.07× respectively, when compared to [11]. In addition to [11], [10] also reports its utilization based on the number of functional units (FUs) that are actively running. Due to the modularized design, the hardware utilization rates during bootstrapping and MNIST with unencrypted weights tasks on [10] are 0.42 and 0.38, respectively, which are also much lower than ours (0.86 and 0.87, resp.).

The above analysis indicates that the improvement of the area-performance efficiency of Alchemist primarily stems from the extraction of Meta-OP and the unified architecture design, which leads to high utilization rates and a decrease in computational overhead.

CONCLUSION

In this work, we propose Alchemist, a unified accelerator architecture that efficiently supports cross-scheme FHE computing. Our key idea is to extract a finer-grained operator called a Meta-OP which mathematically represents diverse polynomial operators. Based on the extracted Meta-OP, our accelerator architecture provides efficient support for both arithmetic and logic FHE computing with the scheduling strategy. The experimental results show that our accelerator achieves a significant speedup compared to existing designs on both schemes.

ACKNOWLEDGEMENT

This work was supported by the National Key R&D Program of China under grant no. 2023YFB3106200, the National Natural Science Foundation of China under grant no. (62090024, 92373206, U20A20202, 62202028), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB44030200), and Huawei Technologies Co., Ltd. The corresponding author is Xing Hu.

REFERENCES

[1] W.-j. Lu *et al.*, "Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data," *Cryptology ePrint Archive*, 2016.
[2] J. Fan *et al.*, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.
[3] J. H. Cheon *et al.*, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*, 2017.
[4] I. Chillotti *et al.*, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, 2020.
[5] C. Boura *et al.*, "Chimera: Combining ring-lwe-based fully homomorphic encryption schemes," *Journal of Mathematical Cryptology*, 2020.
[6] W.-j. Lu *et al.*, "Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *SP*, 2021.
[7] N. Samardzic *et al.*, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO*, 2021.
[8] S. Kim *et al.*, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *ISCA*, 2022.
[9] J. Kim *et al.*, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *MICRO*, 2022.
[10] N. Samardzic *et al.*, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA*, 2022.
[11] J. Kim *et al.*, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *ISCA*, 2023.
[12] M. Knezevic *et al.*, "Faster interleaved modular multiplication based on barrett and montgomery reduction methods," *TC*, 2010.
[13] M. S. Riazi *et al.*, "Heax: An architecture for computing on encrypted data," in *ASPLOS*, 2020.
[14] R. Agrawal *et al.*, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *HPCA*, 2023.
[15] Y. Yang *et al.*, "Poseidon: Practical homomorphic encryption accelerator," in *HPCA*, 2023.
[16] R. Agrawal *et al.*, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *MICRO*, 2023.
[17] L. Jiang *et al.*, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *DAC*, 2022.
[18] A. Putra *et al.*, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," in *MICRO*, 2023.
[19] L. Ding, *et al.*, "Pima-lpn: Processing-in-memory acceleration for efficient lpn-based post-quantum cryptography," in *DAC*, 2023.
[20] W. Jung *et al.*, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *CHES*, 2021.
[21] A. Brutzkus *et al.*, "Low latency privacy preserving inference," in *ICML*, 2019.
[22] I. Chillotti *et al.*, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC*, 2020.
[23] NuCypher, "Nufhe, a gpu-powered torus fhe implementation," https://github.com/nucypher/nufhe, 2020, [Online; accessed November 20, 2023].