2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00088

EFFACT: A Highly Efficient Full-Stack FHE Acceleration Platform

Yi Huang*, Xinsheng Gong*, Xiangyu Kong, Dibei Chen, Jianfeng Zhu†, Wenping Zhu, Liangwei Li,
Mingyu Gao, Shaojun Wei, Aoyang Zhang and Leibo Liu†
Tsinghua University
Equal Contribution*
Corresponding Authors†
{yi-huang,jfzhu,liulb}@tsinghua.edu.cn

***Abstract*—Fully Homomorphic Encryption (FHE) is a set of powerful cryptographic schemes that allows computation to be performed directly on encrypted data with an unlimited depth. Despite FHE's promising in privacy-preserving computing, yet in most FHE schemes, ciphertext generally blows up thousands of times compared to the original message, and the massive amount of data load from off-chip memory for bootstrapping and privacy-preserving machine learning applications (such as HELR, ResNet-20), both degrade the performance of FHE-based computation. Several hardware designs have been proposed to address this issue, however, most of them require enormous resources and power. An acceleration platform with easy programmability, high efficiency, and low overhead is a prerequisite for practical application.**

This paper proposes EFFACT, a highly efficient full-stack FHE acceleration platform with a compiler that provides comprehensive optimizations and vector-friendly hardware. We start by examining the computational overhead across different real-world benchmarks to highlight the potential benefits of reallocating computing resources for efficiency enhancement. Then we make a design space exploration to find an optimal SRAM size with high utilization and low cost. On the other hand, EFFACT features a novel optimization named streaming memory access which is proposed to enable high throughput with limited SRAMs. Regarding the software-side optimization, we also propose a circuit-level function unit reuse scheme, to substantially reduce the computing resources without performance degradation. Moreover, we design novel NTT and automorphism units that are suitable for a cost-sensitive and highly efficient architecture, leading to low area. For generality, EFFACT is also equipped with an ISA and a compiler backend that can support several FHE schemes like CKKS, BGV, and BFV.

We provide both FPGA and ASIC versions of EFFACT. On account of our full stack design, FPGA-EFFACT outperforms the SOTA FPGA accelerators in gmean by 1.22×. Meanwhile, ASIC-EFFACT shows increased improvements in terms of the performance per chip area and the performance per Watt compared with the SOTA ASIC works.

I. Introduction

Fully Homomorphic Encryption (FHE), due to its unique ability to compute encrypted data without requiring a secret key, is indispensable in the privacy-preserving computing field. In certain data privacy-sensitive scenarios like finance and medicine, FHE enables clients to access the computing and storage capabilities of cloud servers without disclosing confidential information.

FHE has been used in many applications to protect user's privacy data, e.g. ACGT base-pairs sequence in the genomic computing [45], the financial transaction [25], and the image processing in CT [22], [42]. However, the amount of computations performed on FHE's encrypted data exceeds that of the unencrypted data. This is mainly because most FHE schemes are lattice-based (e.g., BGV, BFV, CKKS, TFHE) [14], [18], [19], [21], which utilizes the complexity of the Learning With Error (LWE) problem [60] to ensure the security. Besides, FHE schemes necessitate extra operations for ciphertext maintenance to ensure the compactness and correctness of the scheme. These maintenance operations, such as key switching, rescale, and bootstrapping, are not efficient in CPU [76]. Therefore, FHE applications usually need thousands of seconds to be completed on the CPU while the same applications on the unencrypted data only require milliseconds. To tackle this problem, various software methods have been explored, including optimizing FHE algorithms [10], [17], [23] and developing highly optimized HE libraries for CPU or GPU-based systems [8], [16], [24], [49]. However, their performance enhancements remain inadequate to meet the demand of practical applications. In recent years, researchers have focused on creating Domain Specific Architecture (DSA) for FHE to make it more applicable to real-world scenarios.

Currently, efforts have been made to implement ASIC and FPGA designs for FHE. The ASIC designs such as CraterLake [63], BTS [35], and ARK [34] provided more than 2 orders of magnitude speedup over GPU, showing a promising future for the large scale adoption of FHE. However, they have not explored the computation overhead and off-chip bandwidth efficiency. As a result, they require enormous resources, typically hundreds of megabytes of on-chip SRAM and tens of thousands of multipliers. The huge on-chip resources incur large area consumption and huge energy consumption, which is expensive for commercial use and leads to low efficiency [33], [63]. On the contrary, the FPGA solutions [4], [75] are much more efficient due to their highly reused design. However, the FPGA implementations only execute one HE operation every time due to the limited resources, and just a handful of scheduling or parallelism is explored, restricting their throughput. A practical design that features high throughput, flexibility, and high efficiency is demanded in the area of

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00088

commercial FHE acceleration.

MAD [2] and SHARP [33] are the pioneers in designing a cost-sensitive and highly efficient architecture. SHARP gives an in-depth analysis of the word length to tackle the memory and NoC bottleneck, which is orthogonal to our proposal. MAD proposes a novel caching scheme to explore data reuse and reduces the on-chip SRAM requirement by 16×. However, MAD only looks into the on-chip SRAM optimization while still keeping the computing resources, its buffers and SRAM bandwidth as high as prior designs [34], [35], [63], thus still requiring substantial area and energy consumption. Meanwhile, its caching scheme falls into hand-tuned data path scheduling within HE primitives and relies on the buffers on the computing resource side, revealing further optimizations. Therefore, there remains a significant design space for developing a cost-effective and highly efficient FHE accelerator.

To alleviate those drawbacks, in this work, we first exhibit the possibility of re-distributing the computing resources by a detailed analysis of the proportion of different FHE operations at the residue polynomial level in real-world benchmarks. Then we also take a design exploration of the size of on-chip memory based on several trade-offs including performance, efficiency, and cost. Exploring computing resources and on-chip memory to achieve high efficiency and low cost while maintaining high throughput is an essential step in many accelerator designs [37]–[39], [46], [47], [67], [69], [74]. However, such a resource-aware design is not well explored in prior FHE works.

Starting from the analysis, we build EFFACT, a highly **E**fficient **F**ull-stack **F**HE **AC**celeration pla**T**form to enhance the performance, area efficiency, and power efficiency of a cost-sensitive FHE accelerator. To obtain high speed with limited on-chip memory, we propose an automatic software-level optimization named streaming and its architecture-level support, in which the compiler finds the temporary data with less reuse and directly sends them from DRAM to function units without the buffering of SRAMs [53], [65]. To enhance the area and power efficiency, we propose a circuit-level reuse scheme that judiciously uses both the NTT's multipliers and modular mult's multipliers to accelerate mult-accumulate (MAC) operations without performance degradation. Meanwhile, we also devise specialized NTT and automorphism units to adapt to the cost-sensitive and highly efficient microarchitecture, further reducing computing resources.

Without loss of generality, we also analyze the basic residue polynomial level operations in different FHE schemes including CKKS, BGV, and BFV, and design EFFACT's Instruction Set Architecture (ISA) and a compiler backend featuring automatic code optimization. Through the efforts of the ISA and compiler, our platform can flexibly support many kinds of FHE schemes and can be integrated into the state-of-the-art compiler frontends and analysis [44], [54].

The contribution of this work can be summarized as follows:

- We propose EFFACT, a full-stack FHE acceleration platform with generalized ISA and compiler backend, which runs near speed as resource-unconstrained designs with higher efficiency and low hardware overhead.
- We analyze the proportion of different FHE operations among different benchmarks and show the impact of different resource assignments on the efficiency and performance of the FHE accelerator.
- We propose a novel compiler optimization scheme on the software side with its architectural support that keeps high throughput while using extremely small on-chip memory.
- We devise a circuit-level reuse scheme for function units that reduces hardware redundancy without performance penalty and specialized resource-saving NTT and automorphism units.
- We rigorously evaluate EFFACT in logistic regression and bootstrapping with both ASIC and FPGA versions. The experiments are performed by scaling the real FPGA runtime. Results are verified by comparing with Lattigo [49]. On account of our full stack design, FPGA-EFFACT outperforms the SOTA FPGA accelerators in gmean by 1.22×. Meanwhile, ASIC-EFFACT shows increased improvements by ≥1.46× in terms of the performance per chip area and ≥1.48× in terms of the performance per Watt compared with the SOTA ASIC works.

II. Background

As mentioned above, there are different types of FHE schemes. In this section, we will take CKKS as an example to explain how FHE works. Relevant parameters and notations for the CKKS scheme can be found in Table I.

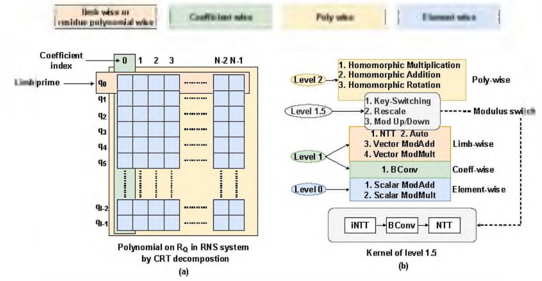
TABLE I
CKKS Scheme Parameter and Notation

Notation	Description
N	Degree of cyclotomic ring
Q	Biggest modulus of ciphertext
P	Modulus product of all extension limbs
R_Q	Cyclotomic polynomial ring, R_Q =$Z_Q[X]/(X^N+1)$
q_i	Modulus at level i of the modulus chain
L	Max level of ciphertext
l	Current level of ciphertext
$dnum$	Number of decompose digits
L_{boot}	Consumed level of bootstrapping
L_{CtS}	Consumed level of CtS in bootstrapping
L_{StC}	Consumed level of StC in bootstrapping
$L_{EvalMod}$	Consumed level of EvalMod in bootstrapping

A. RNS-CKKS FHE scheme

CKKS, proposed by Cheon et al. [18], supports fixed-point real and complex data types and SIMD operation, which introduces approximate calculation into homomorphic encryption algorithm. It trades the loss of accuracy for a huge increase in computational efficiency compared to BGV/BFV schemes.

In EFFACT platform, we ignore the encoding/decoding and encrypting/decrypting steps of the scheme, since these are executed on the client side. Instead, we focus directly on the plaintext, ciphertext, and related homomorphic operations. In CKKS, plaintext can be presented as a polynomial $m(X) = \sum_{i=0}^{N-1} m_i X^i$. The plaintext is an element of the cyclotomic



Polynomial on R_Q in RNS system by CRT decompostion

(a)

Kernel of level 1.5

(b)

Fig. 1. (a)The limb-wise and coefficient-wise data in a polynomial on R_Q with RNS system. (b)The level of HE operations.

polynomial ring $R_Q = Z_Q[X]/(X^N+1)$, which means all the N-degree polynomial plaintext's coefficients (m_i) fall within the range [0, Q-1] and the number of their coefficients is N. Typically, N is the power of 2. A single plaintext is packed (encoded) from a so-called message, which consists of a vector of N/2 complex numbers. Each **element** on the plaintext is called a **slot**. The multiplication or addition of the message can be performed through the polynomial operations on the plaintext. Then the plaintext ($m(X) \in R_Q$) will be encrypted into the ciphertext ($\mathbf{ct}(X) \in R_Q^2$). The ciphertext is represented as $\mathbf{ct}(X) = (c_0(X), c_1(X))$ satisfying $c_0(X) = c_1(X) \cdot s(X) + \Delta m(X) + e(X)$, where $s(X)$ is the secret key, $c_1(X) \in R_Q$ is a random polynomial, $e(X) \in R_Q$ is a small error polynomial is added for security, and Δ is the scaling factor. The recovery of plaintext is conducted by $m'(X) = \mathbf{ct}(X) \cdot (1, -s(X)) = \Delta m(X) + e(X)$.

RNS-CKKS scheme [10], [17], [23] further improves the efficiency by using the Chinese Remainder Theorem (CRT) to decompose a big prime base (the Q base in the R_Q, which requires thousands of bits) into L small primes (denoted as $Q = \prod_{i=0}^{L-1} q_i$) with shorter bit width. Therefore, a polynomial $m(X)$ in the R_Q can be represented using L **residue polynomials** or **limbs** in R_{q_i} as $\{m_0[X] \in R_{q_0}, m_1[X] \in R_{q_1}, ..., m_{L-1}[X] \in R_{q_{L-1}}\}$ or simply $\{[m[X]]_{q_i}\}_{i \in L}$. It not only reduces the bit width of coefficients but also provides the possibility of parallel computation among different residue polynomials in R_{q_i}. All the computations on the polynomial level can be easily extended into the RNS-based residue polynomials.

The level of basic HE operations is shown in Figure 1.b, which is categorized by the types of operands, which are represented as element-wise, residue-polynomial-wise, coefficient-wise, and polynomial-wise as shown in Figure 1.a. For example, given two ciphertexts $\mathbf{ct_0}$= $(a_0 \in R_Q, a_1 \in R_Q)$ and $\mathbf{ct_1}$= $(b_0 \in R_Q, b_1 \in R_Q)$, the Homomorphic Addition (HADD) operates on their polynomials by computing $\mathbf{ct_{add}}$= $(a_0 + b_0, a_1 + b_1)$. When it is broken into residue-polynomial-wise vector Modular Addition (MADD), the two ciphertexts can be represented as $\mathbf{ct_0}$= $(\{[a_0]_{q_i}\}_{i \in L}, \{[a_1]_{q_i}\}_{i \in L})$ and $\mathbf{ct_1}$= $(\{[b_0]_{q_i}\}_{i \in L}, \{[b_1]_{q_i}\}_{i \in L})$. Then the HADD means performing $\mathbf{ct_{add}}$= $(\{[a_0]_{q_i} + [b_0]_{q_i}\}_{i \in L}, \{[b_1]_{q_i} + [a_1]_{q_i}\}_{i \in L})$. The computing kernel of level 1.5 and lower (level 1/0) is of importance in many architectures including EFFACT.

B. Number Theoretic Transformation (NTT)

NTT is a variant of Discrete Fourier Transform (DFT), but it performs computation in a finite field. In CKKS schemes, NTT is used to speed up polynomial or residue polynomial multiplication. Performing polynomial multiplication in the out-of-NTT domain takes $O(N^2)$ asymptotic time, whereas the same operation only requires $O(NlogN)$ asymptotic time in the NTT domain, similar to the way FFT speeds up convolution. To eliminate the cost of padding to 2N-point NTT when performing N-point multiplication, [7], [57] proposes the Negative Wrapped Convolution (NWC) algorithm, which further improves the efficiency of NTT polynomial multiplication. The NWC-based NTT algorithm is defined as follows, let $A(X) = NWC\_NTT(a(X))$, we have:

$$A_j = \sum_{i=0}^{N-1} a_i \omega_{2N}^{(2i+1)j} \quad mod\ Q \tag{1}$$

where A_j is the j-th coefficient of $A(X) \in R_Q$, a_i is the i-th coefficient of $a(X) \in R_Q$, and ω_{2N} is the 2N-th root of prime Q. With the similar algorithmic optimization of FFT [7], the equation can be broken down into a sequence of butterfly operations on R_Q.

Due to the linearity and bit-reversal property of NTT operation [7], [57], many arithmetic characteristics have been maintained between the NTT domain and out-of-NTT domain, we conclude these characteristics of NTT operation as follows:

$$\begin{aligned} NTT(a * b) &= NTT(a) \cdot NTT(b) \\ NTT(a + b) &= NTT(a) + NTT(b) \\ NTT(\sigma_s(a)) &= BR(\sigma'_s(BR(NTT(a)))) \end{aligned} \tag{2}$$

where $a, b \in R_Q$, "$*$" is the convolution signal, "$\cdot$" means residue-polynomial-wise vector production, "+" means residue-polynomial-wise vector addition, "$\sigma_s()$" is the s-step automorphism operation in the out-of-NTT domain, "$\sigma'_s()$" is the s-step automorphism operation in NTT domain, "BR" denotes the bit reversal operations.

C. Key-switching and Base Conversion (BConv)

Homomorphic Multiplication (HMULT) on two ciphertexts $\mathbf{ct_0}$= $(a_0 \in R_Q, a_1 \in R_Q)$ and $\mathbf{ct_1}$= $(b_0 \in R_Q, b_1 \in R_Q)$ requires both polynomial-wise multiplications and a key-switching process. It first computes the intermediate results (d_0, d_1, d_2), where $d_0 = a_0 \cdot b_0$, $d_1 = a_1 \cdot b_0 + a_0 \cdot b_1$, and $d_2 = a_1 \cdot b_1$. The tuple (d_0, d_1, d_2) is decryptable under the secret tuple $(1, s, s^2)$, where s is the secret key. The **evk**= (evk_a, evk_b) is a ciphertext in the ring R_{PQ}^2 with a extended special prime P. The RNS can also be applied to the P base by $P = \prod_{i=0}^{k-1} p_i$. Such an **evk** is different from the input ciphertext's secret key, which is applied in key-switching to change the secret key back to $(1, s)$. The key-switching is the process by computing **ksw**=$P^{-1}(d_2 \cdot$ **evk**), and the final HMULT result is $\mathbf{ct_{mult}}$=(d_0, d_1)+**ksw**.

Base conversion is a major operation in the key-switching. It converts a set of residue polynomials from one modulus set to another set, therefore, it can change the intermediate

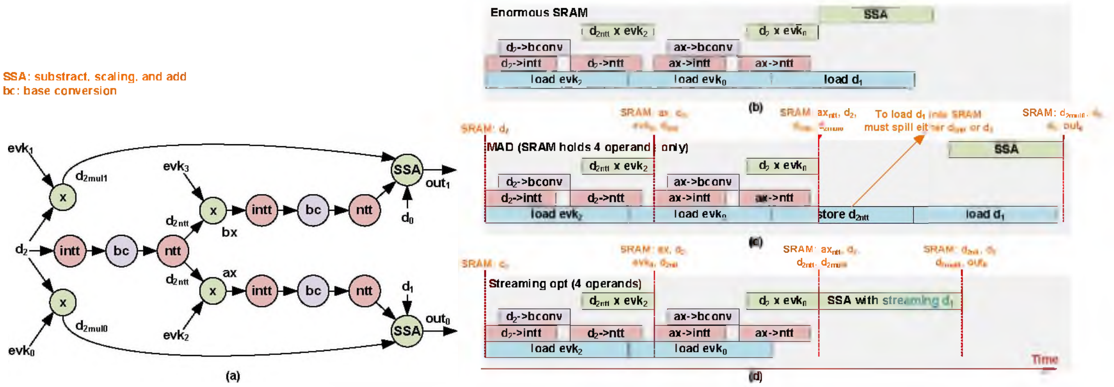


Fig. 2. A toy example of performing key-switching in HMULT. d_0, d_1, and d_2 are the intermediate multiplication results noted in Section II-C, and the evk_0 and evk_2 are evk_a's Q base and P base representations where the evk_a is the component of evaluation key **evk**=(evk_a, evk_b). We only show the timing diagram of the branch below the data flow graph (DFG). (a) Example key-switching DFG, (b) The timing graph of architectures with enormous SRAM and buffers that can hold all temporary operands, (c) The timing graph of MAD with limited SRAM and buffers that can only hold 4 operands, (d) The timing graph of MAD with our streaming optimization, in which we successfully reserve the d_{2ntt} for the reuse in the branch above DFG, reducing extra spills. The latency of streaming optimized instruction is determined by the longest latency of the merged instructions. Here is the latency of loading d_1.

results of HMULT to match the **evk**'s modulus set. For a RNS representation a_C on primes set $C = \{q_0, q_1, ..., q_{l-1}\}$, the fast base conversion from primes set C to $B = \{p_0, p_1, ..., p_{k-1}\}$ is define as follows:

$$BConv_{C \to B}(a_C) = \{(\sum_{j=0}^{l-1} (a_C[j] \cdot \hat{q}_j^{-1})_{q_j} \cdot \hat{q}_j)_{p_i}\} \quad 0 \leq i < k \tag{3}$$

where $\hat{q}_j = \prod_{j' \neq j} q_{j'}$, $(\cdot)_{q_i}$ means reduce the number to Z_{q_i}. BConv operation is widely used in the CKKS scheme, almost as frequent as NTT/iNTT.

D. Prior FHE accelerators

In prior resource-unconstrained FHE accelerator designs, enormous on-chip SRAM and resource-hungry functional units, e.g., base conversion unit and NTT/iNTT unit, are used for tackling the memory-hungry and compute-hungry challenges in FHE computations. Despite their remarkable improvement, the use of hundreds of megabytes of SRAM and fully pipelined functional units makes the area and energy consumption high. Meanwhile, the off-chip memory bandwidth and functional units stay idle most of the time, e.g., less than 50% HBM and ~25% base conversion utilization [33], [63]. Therefore, prior accelerators have a low efficiency with high cost due to the lack of memory and computing efficiency exploration.

SHARP [33] provides an in-depth analysis of the impact of word length of slots on the security level and the data movement, thus, greatly enhancing area and power efficiency by exploring the optimal word length and a NoC-friendly hierarchical architecture. SHARP's insightful proposals can be extended to other accelerators such as ARK [34], and are orthogonal to EFFACT as well. MAD [2] proposed a set of novel caching strategies to explore memory efficiency leading to enormous SRAM reduction. With the O(α) caching data path which only causes load/store of intermediate results at the modulus switching and uses computing resources' buffers to buffer intermediate results in other operations, DRAM transfers are reduced by 44% compared to the naive bootstrapping baseline. However, these caching strategies are still within the category of custom manual adjustments, which only optimize several HE primitives and rely on the buffers on the computing resource side and large numbers of computing resources to hold and consume the intermediate results. There is parallelism unexplored between the HE primitives when given limited SRAMs and buffers. On the other hand, MAD requires large amounts of computing resources to immediately consume their data flow, or severe memory spills and data flow stalls arise. The computation inefficiency problem in the prior architecture is still unsolved. While SHARP and MAD have paved the way for efficient FHE accelerators, the incomplete exploration of memory efficiency and computing resource optimization makes them still fall short of achieving a highly efficient design with low cost.

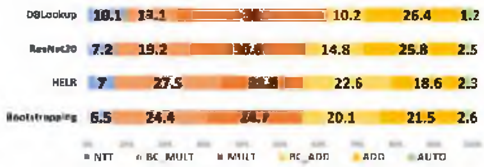


Fig. 3. Residue polynomial level instruction counts in DBLookup, ResNet20, HELR and Bootstrapping. BC_MULT and BC_ADD are MULT and ADD instructions used in BConv, while MULT and ADD represent the normal MULT and ADD except those in BConv.

III. Opportunities and EFFACT proposals

The first opportunity comes from the computing resource side. MAD [2] keeps a similar computing resource distribution as resource unconstrained accelerators to get extreme performance of function units and smoothly conduct their manual data flow without memory spills or pipeline stalls. This causes their computing resources to occupy nearly 90% area of MAD. MAD rarely explores circuit-level function unit reuse and computing resource optimization, which are essential for creating highly efficient and cost-effective designs. Figure 3 shows the analysis of different instructions in IR format of HELR and fully-packed bootstrapping benchmark. The IR program is at the residue polynomial level, where NTT instructions only account for 7% and 6.5% of all instructions. The majority of instructions are MULT and ADD operations, which totally account for 90.7% and 90.9% of all instructions, in which there are 52.7% of MULT instructions and 51.6% of ADD instructions are used for BConv. We call those MULT and ADD not for BConv the normal MULT and ADD.

For the computations of a single ciphertext, we observe that among the normal ADD and MULT instructions (those not for BConv), nearly 77.6% of them cannot be well hidden by the computing of $iNTT - BConv - NTT$ (dnum = 4, a practical parameter widely used in [2], [33], [34]) in previous architectures [34], [63]. Such a situation arises since (1) 77.6% of the normal ADD and MULT instructions, e.g., so-called MatMul1D, BlockMatMul1D in HElib [24], appear behind long $iNTT - BConv - NTT$ chains. Such a pattern also frequently shows up in bootstrapping's CtS/StC, ResNet's convolutions [43], and HELR's gradient calculation [25]. Due to the coefficient-wise aggregation nature of BConv, only the NTT at the end of the last chain or iNTT in the front chain (occupying only 0.88%$\sim$1.75% among all the instructions) can reveal parallelism with normal ADD and MULT instructions behind chains, (2) due to the inevitable data dependency caused by digit-decomposed key-switching algorithm [26] and hoisting rotation algorithm [13] which need aggregation or automorphism in the last step, a majority of data from the NTT at the end of chains cannot be immediately used for subsequent normal multiplication or addition unless they undergo a series of aggregations, further blocking the parallelism. Greatly enhancing the units for normal MULT and ADD instructions may relieve this problem, however, it is out of EFFACT's scope. Based on the above observation, we propose three optimizations as follows:

1) Removing BConv units: Since BConv MULT and ADD instructions cannot execute in parallel with most normal MULT and ADD instructions, we break BConv into a series of vector MULT and ADD instructions in the residue-polynomial-wise execution mode and use the normal MULT and ADD units to execute BConv, removing the specialized BConv units which occupy 66.7% area of all computing units in CraterLake [63]. On the other hand, due to the residue-polynomial-wise execution mode, the computation inefficiency caused by ciphertext level change is also well solved.

2) Reusing function units at circuit level: Since a large number of normal MULT and ADD instructions cannot run in parallel with NTT and iNTT, we devise a circuit-level NTT reuse scheme that leverages reconfigurable technology to intelligently reuse NTT units and modular multiplication units with no performance penalty. The butterfly units' modular multipliers and adders within the NTT units can be reused as MAC units to substitute consecutive normal MULT and ADD instructions, providing speed-up with low hardware overhead.

3) Fine-grained NTT units: Since nearly 17% of the total instructions (see Figure 3) cannot run in parallel with NTT/iNTT, we tend to adopt a fine-grained NTT pipeline similar to [4], [75] instead of the fully-pipelined NTT. Unlike the fully-pipelined NTT unit in which each NTT pipeline stage possesses its own modular multiplier and adder, the fine-grained NTT unit tries to share the same modular multiplier and adder among all the NTT pipeline stages. Therefore, the fully-pipelined NTT [9] runs at a better performance with a larger area than the fine-grained one. However, in resource-constrained scenarios, the fully-pipelined NTT design can easily disrupt the equilibrium between speed and hardware overhead. Using the proportion in Figure 3 and assuming other instructions can be perfectly executed in parallel with NTT and normal MULT and ADD without considering the DRAM fetches, the fully-pipelined NTT design can only achieve $\leq$2.7× performance enhancement compared to the fine-grained NTT design at the cost of $\geq$8× total computing resource consumption. Therefore, implementing such fully-pipelined NTT units is not a better trade-off in the case of a highly efficient and cost-sensitive design.

Another opportunity lies in the on-chip memory side. MAD uses a novel cache scheme to reduce the SRAM cost. However, it does not fully explore the memory efficiency due to its manual optimization and the need for computing resources' distributed buffers used for buffering the intermediate results. They keep prior designs' buffers. In their data flow, data are firstly fetched from DRAM to the SRAM and then flowed to the computing resources. The data will stay between computing resources and their buffers until the computation stage is finished. Such a data path works perfectly when the on-chip SRAM and buffers are large enough to hold all the operands and intermediate results for instructions like in BTS, ARK, and SHARP. However, as shown in Figure 2(c), the reduced SRAM and buffer size causes extra spills and, thus, limit MAD's performance.

To overcome the above challenge, we propose streaming data flow optimization as depicted in Figure 2(d). In our optimization, the function units can get their operands either from the SRAM or directly from the DRAM, which not only reduces the extra stores but also enhances EFFACT's DRAM bandwidth utilization. Our compiler uses a static analysis pass to decide which instructions should get their operands from the DRAM.

Besides the above optimizations, EFFACT also provides an analysis of the interplay among SRAM capacity, bootstrapping running time, off-chip memory bandwidth utilization, and function unit utilization. The compiler of EFFACT automatically explores reuse opportunities and minimizes off-chip data movement at the program level using static scheduling and register allocation. Figure 4 depicts the DRAM bandwidth, function unit utilization, and total runtime variation with different on-chip memory sizes in EFFACT, highlighting the performance and efficiency turning points at 27MB and 54MB. It is worth noting that the MULT and ADD units are almost saturated at $\leq$50% because most of the normal MULT and

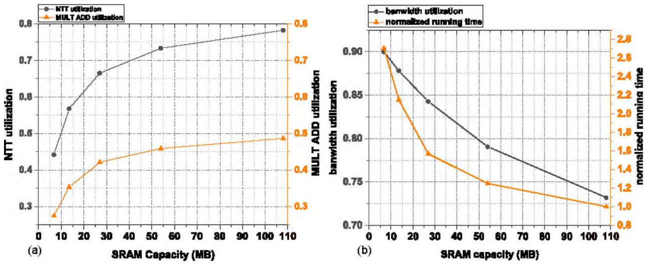


Fig. 4. Impact of different SRAM sizes on utilization and total run time given certain computing resources, we do not show automorphism utilization since it is always low. (a) NTT, and MULT ADD units utilization with different on-chip memory, (b) DRAM bandwidth utilization and total running time with different on-chip memory.

ADD can only be executed serially with (i)NTT except for BConv operations, and in the EFFACT design, (i)NTT will also conduct some of the normal MULT and ADD. We choose 27MB as a trade-off between performance, cost, and efficiency. We also use SimFHE [2] to simulate DRAM transfers of MAD with O(α) caching strategy under identical architectural parameter setting, the result shows that the DRAM transfers of EFFACT are reduced by $\sim$40% compared to MAD.

To make EFFACT more practical and generic, we build a full-stack acceleration platform. Despite EFFACT's hardware design and optimizations, we also extract common residue polynomial level vector instructions from different FHE schemes to develop a generic ISA for EFFACT. At the same time, we also propose a compiler backend with code optimization for EFFACT that can be adapted to the recent compiler frontend [54].

IV. EFFACT Platform

A. ISA overview

EFFACT looks into HE primitives and breaks those primitives at the level of residue polynomials. For generality, we analyze several FHE schemes' operations including CKKS, BGV, and BFV, and thus establish the vector ISA as shown in Table II. For example, the MADD operation means adding one residue polynomial vector to another or a constant by a specific modulus. In the analyzed FHE schemes, operations of level 1 in Figure 1 fall into two categories. One is the residue-polynomial-wise operation in (i)NTT/automorphism or between two polynomials (feeding one polynomial's residue polynomial to the other polynomial's residue polynomial). The other is the coefficient-wise operation within one polynomial. Our ISA can fully support both categories. Our residue-polynomial-wise ISA naturally supports the residue-polynomial-wise operations between two polynomials. Since the coefficient-wise operations' behavior is the same for all the slots in the same residue polynomial, the coefficient-wise operations within one polynomial can be conducted by keeping a residue polynomial of the given polynomial in the memory and then feeding the next residue polynomial to it. In this way, the coefficient-wise operation is changed into the residue polynomial vector operation. In addition, we keep some of the int64 operations (scalar subset) to perform control flow such as loops.

TABLE II
EFFACT ISA

Instruction	Description
MMUL dest src0, src1(imm), modulus	perform modular multiplication on residues
MMAD dest src0, src1(imm), modulus	perform modular addition on residues
(i)NTT dest src0, modulus	perform (i)NTT on a residues
AUTO dest src0, imm, modulus	perform automorphism on a residues
LoadRes dest, srcaddr	load a residue from main memory
StoreRes destaddr, src0	store a residue into main memory
VecCopy dest, src0	move residue among on-chip SRAM
Scalar subset	support loop, branch, and address calculation

B. Compiler Design

Our compiler optimizes the extended LLVM Intermediate Representation (IR) file [41] and generates the executable machine program while also partially supporting control flow features such as loops and branches.

1) Code Optimizations: Our compiler begins by parsing the IR file and the hardware description, lowering the IR instructions to EFFACT's ISA while maintaining the Static Single Assignment (SSA) form. Then it performs copy propagation and constant propagation to eliminate redundant vector copies across different on-chip SRAMs and reduce constant calculation during execution. Our compiler also employs partial redundancy elimination (PRE) using the algorithm described in [15], [32], [36] to eliminate the code redundancy. Additionally, our compiler performs computation merge as mentioned in Section IV-D through a peephole optimization pass. Code optimization is crucial as the automatic IR translator introduces some redundant code. In fully-packed bootstrapping, our code optimizer eliminates 12.9% of instructions, a task that was previously done manually in prior works [2], [33], [34]. Given that previous designs relied on manual optimizations, we have excluded code optimization from our evaluation.

2) Static Scheduling & Memory Allocation: An alias analysis [6] is first performed before scheduling instructions to chain the load/store which may point to the same address in the correct order. Then we schedule the SSA-formed instructions globally as described in [1], [58], [66] to get the optimal parallelism. One big challenge in the compiler is how to manage the on-chip SRAM and the HBM stack to minimize spilling and reduce the load/store when allocating SRAM. Since our vector operation is at the residue polynomial level, we can split the on-chip SRAM into several parts which are the size of one or two residue polynomials, and view each part as a register. Thus, the linear register allocation algorithm [56], [68], [70] can be adopted to allocate on-chip SRAM and manage the HBM.

3) Instruction Merging for Streaming Memory Access: The compiler identifies load operations with a single consumer and merges them as a new streaming operation. The original load operations will not be considered in the succeeding memory allocation phase. The same technique also works in store operations and between different function units.

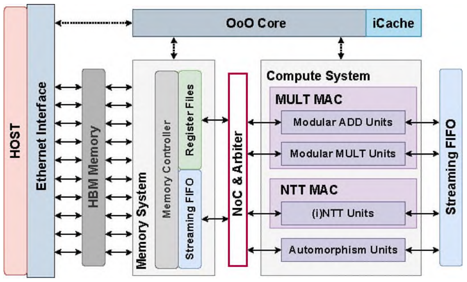


Fig. 5. EFFACT overall hardware architecture.

4) Machine Code Generation: Ultimately, the binary program is obtained by translating the optimized IR into EFFACT's machine code.

C. Streaming Memory Access

In EFFACT architecture, there is a stream memory controller to manage memory accesses. Streaming is a pattern that involves the continuous flow of data through computing units and DRAM. EFFACT employs partial streaming memory access [53], [65], in which part of data can flow directly from DRAM to function units upon arrival instead of waiting for the SRAM chunk to be filled. It aims to maximize computing resource utilization, reduce memory latencies, and more essentially, relieve the SRAM pressure. For stream implementation, a separate FIFO address space is added to efficiently collect data. The memory controller handles concurrent accesses from HBM to the on-chip SRAM or among on-chip SRAM and the aforementioned FIFO space. High parallelism with low overhead can be achieved by fully utilizing the associated resources through arbitration of memory accesses.

As mentioned in Section IV-B, load operations or normal functional operations with only one consumer will be processed with instruction merging and are allowed to use streaming memory access. The merged instruction will first issue the first operation it merges. Through streaming, vector data that is returned by the first operation can be directly read into the aforementioned FIFO space one by one and subsequently dispatched straight to the second operation's FU or the DRAM. If there are no other data dependencies, the second operation will execute upon arrival of the data without needing to wait for the preceding operation to complete. The data is processed on arrival, eliminating the requirement of waiting for the on-chip SRAM to fill up, and enabling nearly real-time processing of data, as shown in Figure 2(d). Otherwise, in the case where data will be used by multiple consumers, it is filled into the on-chip SRAM for reuse.

D. EFFACT Architecture

EFFACT architecture consists of an Out-of-Order (OoO) control core, four functional units (ModAdd, ModMult, NTT, and Auto), FIFOs, register files (RFs, implemented using SRAM), and the interface with off-chip HBM. This section will introduce how these components work in our architecture in Figure 5.

1) OoO core: Both the SRAM and streaming FIFO will request data from DRAM. To fully utilize the DRAM bandwidth, we allow the streaming FIFO and SRAM to compete for DRAM transferring. Since the instruction merge may chain load/store with the fine-grained NTT which is slower than DRAM transferring, tying DRAM to NTT will lead to underutilization of DRAM bandwidth. Therefore, competing for DRAM transferring becomes necessary. To manage the competition of DRAM requests, we introduce an OoO controller that issues instructions when function units are temporarily available and controls in-flight instructions. We implement an OoO core using the scoreboard to track the dependencies and availability of instructions. Since memory ordering has been enforced by the compiler, the results of instructions are ensured to be written back to the register file or the memory in the correct order.

2) Memory and Interfaces: The on-chip SRAM is connected to a multi-channel off-chip HBM for staging data in order to enable increased data reuse and reduced access overhead, as mentioned in Section IV-C. Also, the partial streaming memory access scheme allows direct communication between the function units and HBM to save the step of filling the on-chip SRAM. A memory controller and an arbiter are responsible for managing the data access between function units and the SRAM via banks. An HBM controller manages the load/store operations of the HBM.

Since the SRAM and computing resources are largely decreased thanks to the proposed streaming and hardware optimizations, it alleviates the stringent requirement for large on-chip SRAM bandwidth. Therefore, we are able to reduce the on-chip bandwidth to further reduce the area and energy consumption. Unlike MAD, ARK, and CraterLake, which require at least 90TB/s on-chip SRAM bandwidth, the streaming access and smart function unit reuse scheme relieves the pressure on the SRAM ports and thus, EFFACT only requires about 30TB/s on-chip SRAM bandwidth, giving a roughly threefold improvement.

3) Reconfigurable NTT units: The baseline fine-grained NTT architecture of EFFACT is similar to [4]. However, EFFACT makes several improvements to better adapt to a vector accelerator. First, EFFACT employs the CG-NTT [11] algorithm to implement the fine-grained NTT units due to their vector-friendly characteristics instead of the Cooley-Tukey in [4]. Second, we diminish the bit-reversal operation of each coefficient vector. We perform the bit-reversal operation on twiddle factors rather than the N coefficients since the bit-reversal operation of twiddle factors (TFs) can be done with much less cost during TFs seed pre-computation and TF on-the-fly generation. Moreover, we notice that some modular multiplications and additions cannot run in parallel with the NTT and the NTT naturally possesses a mult-accumulate (MAC) data path in its circuit implementation as shown in the middle of Figure 6. Therefore, we propose to reuse the NTT as a MAC unit while [4] cannot.

Figure 6 shows a sequence of $iNTT - BConv - NTT$ followed by normal MULT and ADD, which shows up frequently

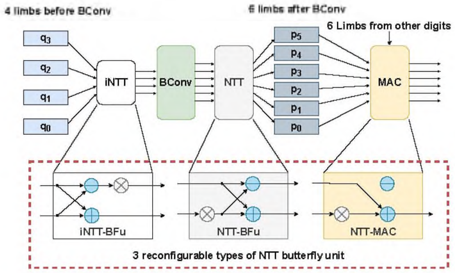


Fig. 6. NTT reuse scheme to accelerate MAC operation.

in operations such as matrix multiplications, convolutions, and hoisting rotations. Based on the observations proposed in Section III, EFFACT reuses NTT unit under following execution modes. First, the butterfly units at the bottom of Figure 6 can be reused as both NTT-butterfly units and iNTT-butterfly units (the middle and left) just like ARK [34] by adding several multiplexers to change the position of the multiplier in the data path. Second, EFFACT also supports configuring the NTT data path as the MAC units to accelerate those MAC operations that can not run in parallel with NTT. It is implemented by simply masking the subtraction out of the data path and picking the MAC result out of the original NTT pipeline. Through this reconfigurable technology, EFFACT gets performance enhancement in MAC operations without much hardware overhead.

4) Auto units: Automorphism is a sub-operation of HROT, where the automorphism with s-step is defined as $\sigma_s(\cdot)$. It can be regarded as a straightforward process of address mapping and sign transformation. In address mapping, the new index of position i for rotation step s is calculated as follows:

$$index_{new}(i,\ s) = i * 5^s \ mod\ N \tag{4}$$

Inspired by ARK [34], we discovered that data in each row remains in the same row after the automorphism operation. This allows us to decompose the 2^{16}-element permutation into N_L-element permutations inside each row and design the auto-mapping units with sign transformation, where N_L is the number of lanes in EFFACT. Such units have also been used in ARK [34] to vectorize automorphism.

However, for the transpose before and after the auto-mapping units in Figure 7(a), SHARP [33] and ARK [34] transpose the input matrix by accessing different rows of their register files simultaneously, which requires their RFs heavily banked. This is not suitable for a vectorized accelerator like EFFACT which can only access SRAM row by row. On the other side, the transpose units proposed by CraterLake [63] demand excessive global connections, causing a large area. Therefore, EFFACT proposes a modified algorithm replacing the expensive matrix transpose operation with a fixed network (FN). We notice when the coefficients in the NTT domain follow a bit-reversal order, each row of the coefficients matrix shares the same transform pattern to turn to the transposed matrix as shown in Figure 7(b), while the transformation in the column can be conducted by changing the SRAM fetching order. This implies that we can simply leverage a fixed network to implement the comprehensive transposed operation.

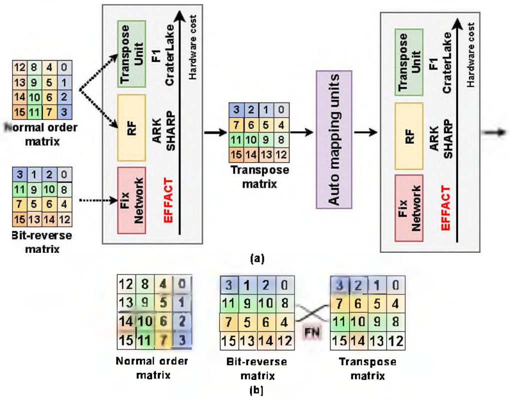


Fig. 7. (a) Hardware structure of different automorphism units and hardware cost comparison, (b) An example of how bit-reverse character works on the 16-point matrix transposition.

5) Merge computation into BConv: Prior works [34], [62] uses the Montgomery modular multiplier due to the relatively lower hardware overhead. However, it will bring extra Montgomery representation transformation penalties when dealing with the modulus switching operations. Meanwhile, the post-processing of iNTT will also bring one extra modular multiplication with a constant. To eliminate both issues, EFFACT analyze the widely-used $iNTT-BConv-NTT$ computation flow and propose to merge these extra computations into the BConv.

In classic Montgomery algorithm [55], input data will be converted into their single-Montgomery (SM) representations like $X \rightarrow XR\ mod\ Q$. The Montgomery modular multiplication is computed on SM representations of input data by $MontMult(XR,\ YR,\ Q) = XYR\ mod\ Q$, which maintains the same representation between the input and output data. The data in EFFACT will be maintained in their SM representations through the whole process when no modulus transformation happens. However, operations like rescale and key-switching require data in the non-Montgomery (NM) format to perform modulus transformation, causing extra Montgomery representation transformation penalties. To reduce the extra Montgomery transformations, we propose a double-Montgomery (DM) representation of constant numbers defined as $X \rightarrow XR^2\ mod\ Q$. Then the DM representation can help to merge the Montgomery transformation into BConv as follows.

Initially, the $(\hat{q_j}^{-1})_{q_j}$ in equation 3 is kept in the NM representation. The input data $a_C[j]$ in equation 3 naturally stays at the SM representation. Therefore, the Montgomery multiplication result of $a_C[j]$ and $(\hat{q_j}^{-1})_{q_j}$ will be in the NM representation. To this end, we pre-compute the constant $(\hat{q_j})_{p_i}$ in equation 3 in the DM representation. The Montgomery multiplication results with $(\hat{q_j})_{p_i}$ in the DM representation will directly change the NM-represented intermediate result from the last multiplication back into the SM representation. With

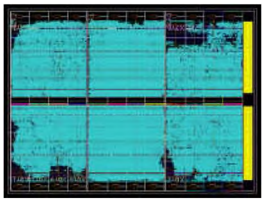


Fig. 8. FPGA layout of EFFACT with 64 lanes.

this optimization, we eliminate the Montgomery representation transformation in the modulus transformation.

On the other hand, a final constant multiplication with $\frac{1}{N}$ is needed when finishing iNTT butterfly operations. In the $iNTT - BConv - NTT$ computation flow, each iNTT operation is followed by a BConv operation, giving the opportunity to merge the $\frac{1}{N}$ constant multiplication into the first constant multiplication of BConv operation, i.e. rewrite the constant $(\hat{q_j}^{-1})_{q_j}$ in equation 3 as $(\hat{q_j}^{-1} * \frac{1}{N})_{q_j}$ which can be pre-computed.

In total, the computation merge that eliminates both the post-processing of iNTT and the Montgomery representation transformation penalties can be solved by redefining the right side of BConv in equation 3 as:

$$\{(\sum_{j=0}^{l-1}(a_C[j]^{SM} \cdot (\hat{q_j}^{-1} * \frac{1}{N})^{NM})_{q_j}) \cdot \hat{q_j}^{DM})_{p_i}\}_{0\leq i<k} \quad (5)$$

V. Experimental Methodology

We evaluate a complete EFFACT system: the compiler is built using C++ with LLVM-like IR and passes, and the microarchitecture is fully implemented in RTL and synthesized by using LVT TSMC 28nm technology node by Synopsis Design Compiler with a commercial SRAM IP [71] licensed by TSMC. Moreover, we further synthesized our RTL design to the Xilinx VCU128 evaluation board using Vivado 2021.1.

A. Benchmarks and Parameters

Fully-packed bootstrapping: Bootstrapping is the pivotal operation of the FHE schemes, we compare the bootstrapping performance of EFFACT with the SOTA GPU [30], FPGA [4], [75], and ASIC [34], [35], [63] implementation. Our fully-packed bootstrapping algorithm consumes $L_{boot} = 15$ level, which includes 4 levels for the CtS procedure, 3 levels for the StC procedure, and 8 levels for the EvalMod procedure.

Logistic regression: Logistic regression is a general model in machine learning which used for binary classification. Previous work HELR [25] proposed a highly efficient logistic regression algorithm based on the CKKS scheme, we regard this work as the baseline, and target at the training phase. The experimental results show that after 30 iterations of training in EFFACT, the accuracy of the inference phase can reach 96.67%.

ResNet-20: ResNet-20 [43] implemented the ResNet-20 DNN model based on the CKKS scheme, which consists of numbers of bootstrapping and homomorphic convolutions, we evaluate the performance of the inference on a single encrypted image as in the original implementation.

DB-lookup based on BGV: As mentioned earlier, EFFACT is a universal acceleration platform for BGV, BFV, and CKKS. We also evaluate the performance of the DB Lookup application proposed in HElib [24] based on the BGV scheme, and we take F1's implementation as our baseline.

In order to show it more intuitively, we list the parameters used in the fully-packed and 256-slot bootstrapping in Table III. HELR starts computations at level 23 and performs 256-slot bootstrapping per two iterations, The parameters of other benchmarks are the same as their original implementations.

TABLE III
Bootstrapping Parameters

$\#Slots$	N	L	L_{boot}	L_{CtS}	$L_{EvalMod}$	L_{StC}	$log(q)$	$dnum$
2^{15}	2^{16}	24	15	4	8	3	54	4
2^{8}	2^{16}	24	13	3	8	2	54	4

B. Compiler

EFFACT's compiler passes are LLVM-style and are in line with the EFFACT ISA. We implemented passes for code optimization, memory ordering, and static scheduling. We also implemented code generation for the EFFACT microarchitecture. The compiler is run on a workstation with 4 Intel Xeon 3.40 GHz CPUs.

C. System-Level Integration & Experimental Platform

EFFACT is implemented completely in RTL, including an OoO core and four function units, together with a User Datagram Protocol (UDP) unit. We provide an ASIC version featured with 1.2-TB/s HBM bandwidth and 1024 lanes which is synthesized without the UDP unit and HBM controller. The power and area of HBM are estimated using [27]. We also implemented the RTL on the Xilinx VCU128 evaluation board as EFFACT's experimental platform.

In the FPGA experimental platform, the evaluation board is connected to the PC through a 1000-Mbit Ethernet interface. We developed a C++ source code based on the socket library to communicate with the FPGA using the UDP. Data for each application begins off-chip and is loaded from FPGA HBM. On the FPGA side, there is a corresponding module that unpacks the UDP-formatted data and deposits it sequentially into the HBM, and the results of the FHE program are also packed up and transmitted using the UDP. Furthermore, the functionality of EFFACT has been thoroughly verified by comparing it with Lattigo [49].

Figure 8 demonstrates the layout of our design. Worth mentioning that the system in FPGA runs only at 12.5 MHz with 64 lanes, although we successfully synthesized it at 300 MHz with 256 lanes on Vivado. The major bottleneck lies in the routing congestion in which the congestion level reaches 7. The HBM bandwidth is also lowered through an asynchronous FIFO to ensure that we can correctly scale the performance of the 12.5 MHz with 64 lanes version to our target 300 MHz with 256 lanes FPGA-EFFACT and 1024 lanes ASIC-EFFACT.

VI. Evaluation

A. Area and Power Analysis

TABLE IV
ASIC-EFFACT breakdown

Components	Area(mm^2)	Power(W)
NTTU	37.13	21.16
MADDU	3.59	3.51
MMULU	18.21	10.12
AUTOU	4.65	4.88
SRAM	81.50	43.14
HBM	29.60	31.80
Others	37.20	21.13

Table IV shows the area and power breakdown of ASIC-EFFACT. We adopt a fine-grained NTT unit in contrast to other ASIC designs and there are no additional transpose units and twisting units in the data path, which leads to only 11% area of the NTTU in ARK. Unlike other designs that use more than 200-MB SRAM, ASIC-EFFACT only requires 27 MB, dropping the SRAM area to 6% compared to ARK. In total, the SRAM occupies 38.46% area and 31.79% power with 30% and 29.22% going to FUs.

TABLE V
ASIC Resource Comparison

	Tech	Freq(GHz)	Area(mm^2)	Power(W)
F1	14/12 nm	1-2	151.4	180.4
BTS	7 nm	0.3-1.2	373.6	133.8
CraterLake	14/12 nm	1-2	472.3	320.0
ARK	7 nm	1	418.3	281.3
CL+MAD-32	14/12 nm	1	333.9	213.4
ASIC-EFFACT	28 nm	0.5	211.9	135.7

Table V shows the resource comparison with recent ASIC designs. By technology scaling [51], [72], [73] (HBM keeps unchanged when scaling), ASIC-EFFACT only requires 0.783×, 0.153×, 0.257×, 0.137×, and 0.414× area consumption compared to F1 [62], BTS [35], CraterLake [63], ARK [34], and CL+MAD-32 [2]. Meanwhile, ASIC-EFFACT nearly achieves the lowest absolute power among these designs. The main area and power drop come from the significant reduction in SRAM capacity (nearly identical to MAD, 2× compared to F1, more than 8× in others), no computing resources' buffers (some necessary pipeline registers in which one register is only lanes×1 large) and the smart function unit reuse scheme. In the next subsection, we will demonstrate that ASIC-EFFACT reaches a higher performance density [48] and power efficiency than prior ASIC designs even though we reduce the function units and SRAM capacity.

TABLE VI
FPGA Resource Comparison

Work	Platform	LUT	FF	BRAM	URAM	DSP
FAB	Xilinx U280	899K	2073K	3840	960	5120
Posidon	Xilinx U280	728K	915K	2048	-	8640
FPGA-EFFACT	Xilinx VCU128	1246K	2096K	1343	864	8212

Table VI lists the resource utilization of our FPGA-EFFACT. Although FPGA-EFFACT only requires 7.6-MB SRAM, the BRAM and URAM utilization reaches more than 50% in the VCU128. This is because the BRAM and URAM array in the FPGA have a depth of 1024 and 4096, in which our residue polynomial mapping only uses 256 rows leading to more than 75% rows unused. Meanwhile, to route FPGA-EFFACT, we use the routability strategy of Vivado which increases our LUT usage from ~900K by default to 1246K.

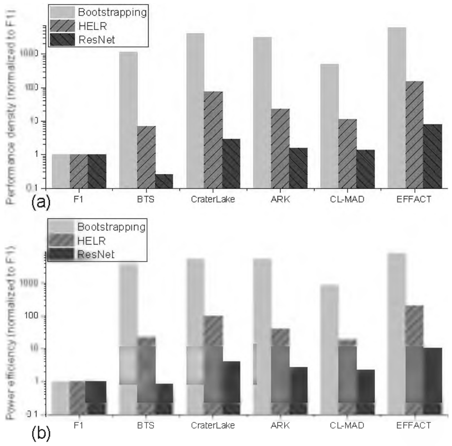


Fig. 9. (a) Performance density comparison, (b) Power efficiency comparison.

B. Performance and Efficiency

We use the amortized time ($T_{A.S}$) [30] to evaluate the performance of Fully-packed Bootstrapping. The amortized time of Bootstrapping effectively captures the reciprocal throughput of a CKKS scheme with a certain parameter set. Therefore, it is widely used to evaluate the Bootstrapping performance in prior FHE accelerators [3], [34], [35]. ASIC-EFFACT is 13.49×, 4743.79×, 0.82×, 0.31×, 0.26×, and 4.93× faster than GPU [30], F1 [62], BTS [35], CraterLake [63], ARK [34], and MAD [2] as shown in Table VII in CKKS. ASIC-EFFACT is slower than most ASIC designs in Bootstrapping since (1) ASIC-EFFACT works only at 500 MHz with fewer multipliers while others run at more than 1 GHz. (2) Bootstrapping features frequent data movement that blocks the parallelism. However, ASIC-EFFACT speeds up over MAD due to (1) our streaming optimization and global memory management reduce nearly 40% DRAM access and the corresponding SRAM latency, (2) excessive static and dynamic scheduling well explores the instruction parallelism compared to MAD's hand-tuned data path, and (3) circuit-level reuse scheme is performed on the highly serial data path which results in acceleration with fewer computing resources. While in HELR, ASIC-EFFACT is 89.1×, 117.7×, 3.26×, 0.43×, 0.89×, and 5.5× faster than GPU, F1, BTS, CraterLake, ARK, and MAD. Unlike Bootstrapping, ASIC-EFFACT becomes more comparable with prior ASIC designs even if fewer function units and lower frequency are provided because HELR does not require as intermediate load/store as Bootstrapping as profiled by CraterLake [63]. The same thing also happens in ResNet, in which ASIC-EFFACT is 6.16×, 4.62×, 0.57×, 0.67×, and 2.35× faster than F1, BTS, CraterLake, ARK, and MAD.

Performance density is calculated by throughput per area to provide an area efficiency comparison and a scale-out evaluation. We scale prior ASIC designs to a 28-nm technology using the scaling technique provided by TSMC in [51], [72], [73], in which we use our SRAM IP to evaluate the

TABLE VII
PERFORMANCE ON BENCHMARKS

	F1/F1+	BTS-2	CraterLake	ARK	FAB	Poseidon	Over 100x	CL+MAD-32	**FPGA-EFFACT**	**ASIC-EFFACT**
Parallelism	2048	2048	2048	1024	256	256	-	2048	256	1024
Multiplier Number	18432	8192	>33792	20480	256	256	-	14336	**512**	**2048**
HBM Bandwidth	1 TB/s	1 TB/s	1 TB/s	1 TB/s	460 GB/s	460 GB/s	-	1 TB/s	460 GB/s	1.2 TB/s
On-chip Memory Cap	64 MB	512 MB	282 MB	588 MB	43 MB	8.6 MB	-	32 MB	**7.6 MB**	**27 MB**
Bootstrapping($T_{A.S.}$)	260 us	0.045 us	0.017 us	0.014 us	0.477 us	0.840 us	0.74 us	0.270us	**0.566 us**	**0.0548 us**
HELR(1 iteration)	1024 ms	28.4 ms	3.73 ms	7.72 ms	103 ms	86.3 ms[1]	775 ms	47.81 ms	**64.55 ms**	**8.7 ms**
ResNet-20	2693 ms	2020 ms	249.45 ms	294 ms	-	2661.23 ms	-	1015.8 ms[2]	**2175.41 ms**	**436.95 ms**
DBLookup	4.36 ms	-	-	-	-	-	-	-	**0.86 ms**	**0.13 ms**

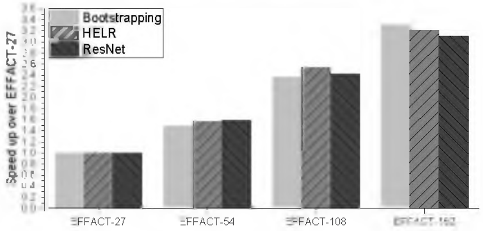


Fig. 10. How performance scales up with memory and computing resources.

SRAM of CraterLake like SHARP [33] does since CraterLake has not reported the exact power of SRAM. Figure 9(a) shows that in terms of performance density, ASIC-EFFACT achieves 6054.9×, 5.35×, 1.46×, 1.86×, and 11.89× higher performance density than F1, BTS, CraterLake, ARK, and MAD in Bootstrapping. While in the HELR and ResNet, ASIC-EFFACT outperforms prior designs at least by 2.02× and 2.7×. We also look into the power efficiency evaluated by performance per Watt. Figure 9(b) depicts that ASIC-EFFACT surpasses F1, BTS, CraterLake, ARK, and MAD by 8256.62×, 2.28×, 1.48×, 1.49×, and 9.76× respectively in Bootstrapping by scaling technology node. ASIC-EFFACT also outperforms prior architectures in HELR and ResNet by at least 2.04× and 2.72×. ASIC-EFFACT achieves the best area and power efficiency due to (1) streaming access removing nearly 40% DRAM fetches compared to MAD, thus, enhancing the SRAM efficiency, (2) computing resources re-assignment based on our analysis, and (3) efficient fine-grained function units and circuit-level reuse scheme further dramatically reducing the area without heavy performance penalty. When applying SHARP's optimization [33] to EFFACT, we find that SAHRP-EFFACT will further improve EFFACT's area and power efficiency by 1.82× and 1.74× on average.

On the FPGA side, FPGA-EFFACT can well explore the parallelism between all the HE operations globally, thus, FPGA-EFFACT boosts both FAB [4] and Poseidon [75] in HELR by 1.59× and 1.34×. We also observe that FPGA-EFFACT outperforms Poseidon in Bootstrapping by 1.48×. However, 7.6-MB SRAM makes the frequent data movement during Bootstrapping worse for our compiler, thus, FPGA-EFFACT cannot exceed FAB in Bootstrapping.

C. Scalibility of EFFACT

To analyze how the computing resources and SRAM capacity impact EFFACT's performance, we create EFFACT-54 with 54-MB SRAM and 4096 multipliers, EFFACT-108 with 108-MB SRAM and 8192 multipliers, and EFFACT-162 with 162-MB SRAM and 12288 multipliers. The performance is evaluated using a cycle-accurate C++ simulator. Figure 10 shows the results. When both the computing and SRAM resources scale up, EFFACT can directly have better performance since (1) when adding multipliers, the performance of NTT instructions increases linearly, (2) the serial normal MULT and ADD can be faster with the help of NTT units, and (3) larger SRAM incur fewer DRAM fetches, which means fewer memory stalls and more DRAM bandwidth can be provided to streaming optimization. According to the simulation results, EFFACT-108 can outperform ARK and CraterLake in HELR and ResNet. Since Bootstrapping is more memory intensive, EFFACT should have a 162-MB SRAM and 12288 multipliers to catch up with ARK and CraterLake.

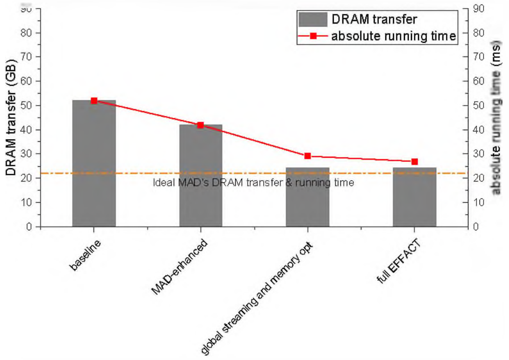


Fig. 11. How Bootstrapping's DRAM transfer and absolute running time change when incrementally applying MAD's optimizations, EFFACT's global scheduling and streaming, and EFFACT's circuit reuse scheme.

D. Application to other schemes

Since the CKKS, BGV, and BFV share the same algebraic structure at the bottom layer, correspondingly, the same basic operations of the ciphertext (ModMult, ModAdd, NTT, Auto), it's obvious that EFFACT is also capable of accelerating the other two schemes, we evaluate ASIC-EFFACT and FPGA-EFFACT over BGV application, DBLookup. From Table VII, ASIC-EFFACT and FPGA-EFFACT are 33.54× and 5.07× faster than F1. As for the boolean FHE scheme, take TFHE [28] as an example, EFFACT also demonstrates excellent acceleration capabilities. The key operation in TFHE is bootstrapping, which contains 3 major sub-operations, $ModulusSwitching$, $BlindRotation$, and $SampleExtraction$. $ModulusSwitching$ can be mapped into modular arithmetic and NTT in EFFACT, even though FFT is used in previous work [29]. For $BlindRotation$ and $SampleExtraction$, as described in [3], [20], when excluding the modular arithmetic and NTT, they are mainly linear shift operations with some slots being reversed. Therefore, EFFACT can support them using our automorphism unit by bypassing the fixed network and controlling the Muxes and reverse units to form shift operations and reverse data. We evaluate ASIC-EFFACT on TFHE Bootstrapping [20] under $N = 2^{13}, logQ = 218, h = 1, l = 2$ like HEAP [3], and it shows the performance of 0.576-ms. Due to the flexibility of supporting different schemes, EFFACT has more potential than prior designs in the transciphering applications such as switching to AES [5] or TFHE [3].

E. Sensitivity Study

Since Bootstrapping operations represent a fundamental HE primitive, we analyze how our optimizations influence Bootstrapping's DRAM transfer and absolute running time to study how EFFACT enhances efficiency in a resource-constrained scenario. We create a bold baseline accelerator under our resource-constrained hardware settings and parameter set similar to ASIC-EFFACT (27-MB SRAM, 1TB/s DRAM bandwidth for simplification, 2048 modular multipliers, and 3072 modular adders) without any optimization and incrementally switch to MAD's and our optimizations. Figure 11 shows the results.

We assume that the baseline, MAD-enhanced baseline, and ideal MAD have an ideal parallelism between memory operations and computations. The ideal MAD has unbounded computing resources and uses its infinite modular multipliers, modular adders and their buffers to support the smooth data flow processing and storing intermediate results, therefore no intermediate results are spilled and its main DRAM transfer is used to load secret keys or perform data structure transformation. However, MAD only looks into several data paths and highly relies on the number of computing resources and their buffers to consume the data flow results immediately. Therefore, with a restricted number of computing resources and their registers, MAD-enhanced baseline cannot perform as well as the ideal MAD. It only reduces the DRAM transfer and absolute running time by 1.24× compared to the non-optimized baseline. Our automatic scheduling and streaming optimization is applied at the whole program level and is aware of the limited computing resources and the small on-chip memory, therefore it not only reduces the 42.2% DRAM transfer but also reduces 30.6% of the absolute running time. When further applying our circuit-level NTT reuse scheme, it does not impact the DRAM transfer since it is only a computing optimization. Instead, it improves the normal ADD and MULT throughput which are shown in Figure 3, leading to a 1.1× absolute running time improvement.

VII. Related Work

CPU/GPU Acceleration. Previous designs have taken a deep dive into the acceleration of HE primitives by making better use of the CPUs and GPUs. Many software libraries including Lattigo [49], HElib [24], SEAL [16], HEAAN [31], and PALISADE [40] have been proposed to improve the HE performance on CPUs. Intel HEXL [12] further used the AVX-512 to accelerate HE operations over SEAL and PALISADE. However, due to the limited resources, CPU schemes remain impractical. Over 100× [30] accelerated the FHE primitives including CKKS bootstrapping on GPUs by better utilizing the memory bandwidth.

FPGA/ASIC Acceleration. HEAX [61] and [64] proposed FPGA solutions for level HE. FAB [4] and Poseidon [75] are pioneers in realizing the fully-packed bootstrapping on FPGA, while HEAP [3] explored the possibility of FPGA acceleration for HE scheme-switching. However, their designs either target leveled HE or never explore the parallelism between HE ops. CoFHEE [50], F1 [62], BTS [35], ARK [34], Cheetah [59], SHARP [33], and CraterLake [63] are ASIC designs to accelerate HE schemes. Cheetah targets privacy-preserving ML using LHE and requires expensive communication overhead. F1, BTS, ARK, SHARP, and CraterLake support fully-packed bootstrapping and show impressive acceleration over GPUs and FPGAs. However, they require a huge amount of on-chip SRAMs and computing resources. CiFlow [52] proposes a novel data flow framework tuned for the hybrid key switch and shows huge data movement reduction. However, their data flow is manually tuned only for the evaluation key's DRAM loading in the key-switching. It highly relies on the large number of computing resources to immediately consume their intermediate results of their data flow.

VIII. Conclusion

In this work, we propose EFFACT, aiming at the challenging problems of highly efficient and cost-sensitive FHE acceleration and propose. EFFACT analyzes the proportion of different FHE operations and the inherent parallelism within them in several real-world benchmarks, exhibiting opportunities to reassign the computing resources. Based on this observation, we tailor the specialized BConv unit, devise novel compact NTT and automorphism units, and propose a circuit-level reuse scheme. These innovations aim to leverage reconfigurable technology to enhance efficiency while minimizing performance penalties. On-chip memory is also significant for a compact accelerator. To fully utilize the limited SRAM, we propose a streaming optimization in which function units can directly fetch their operands from DRAM instead of waiting for SRAM to be filled. Also, we perform a design space exploration to find an optimal SRAM size targeting high efficiency and low cost. Without loss of generality, we also devise a generalized ISA and a compiler backend that can support several FHE schemes and can be integrated into recent compiler frontends. Experimental results demonstrate that EFFACT outperforms state-of-the-art baseline FHE accelerators in terms of efficiency, and area/on-chip memory overhead.

[1] To complete the full HELR benchmark, Poseidon will need more bootstrapping operations rather than 10 iterations combined with 2 bootstrapping.

[2] We evaluate the performance of CL+MAD-32 on ResNet-20 using SimFHE under the same parameter settings as EFFACT.

Acknowledgements

This work is supported in part by the National Key R&D Program of China (Grant No. 2023YFB4403500), and in part by the National Natural Science Foundation of China (Grant No. 62274102)

References

[1] S. Abraham, W. Meleis, and I. Baev, "Efficient backtracking instruction schedulers," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pp. 301–308, ISSN: 1089-795X.

[2] R. Agrawal, L. d. Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 685–697.

[3] R. Agrawal, A. Chandrakasan, and A. Joshi, "Heap: A fully homomorphic encryption accelerator with parallelized bootstrapping," 2024, https://bu-icsg.github.io/publications/2024/fhe_parallelized_bootstrapping_isca_2024.pdf. [Online]. Available: https://bu-icsg.github.io/publications/2024/fhe_parallelized_bootstrapping_isca_2024.pdf

[4] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption," version: 1. [Online]. Available: http://arxiv.org/abs/2207.11872

[5] E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu, "E2E near-standard and practical authenticated transciphering," Cryptology ePrint Archive, Paper 2023/1040, 2023. [Online]. Available: https://eprint.iacr.org/2023/1040

[6] L. O. Andersen and P. Lee, "Program analysis and specialization for the c programming language," 2005.

[7] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient fpga implementations of lattice-based cryptography," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 81–86.

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

[9] D. H. Bailey, "Ffts in external or hierarchical memory," in *Supercomputing '89:Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, 1989, pp. 234–242.

[10] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full RNS variant of FV like somewhat homomorphic encryption schemes," in *Selected Areas in Cryptography – SAC 2016*, R. Avanzi and H. Heys, Eds. Springer International Publishing, vol. 10532, pp. 423–442, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-319-69453-5_23

[11] U. Banerjee, A. Pathak, and A. P. Chandrakasan, "2.3 an energy-efficient configurable lattice cryptography processor for the quantum-secure internet of things," in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2019, pp. 46–48.

[12] F. Boemer, S. Kim, G. Seifu, F. D.M. de Souza, and V. Gopal, "Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 57–62. [Online]. Available: https://doi.org/10.1145/3474366.3486926

[13] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," Cryptology ePrint Archive, Paper 2020/1203, 2020, https://eprint.iacr.org/2020/1203. [Online]. Available: https://eprint.iacr.org/2020/1203

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," p. 35.

[15] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.

[16] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple encrypted arithmetic library v2.3.0," p. 35.

[17] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Selected Areas in Cryptography – SAC 2018*, C. Cid and M. J. Jacobson, Eds. Springer International Publishing, vol. 11349, pp. 347–368, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-10970-7_16

[18] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers." [Online]. Available: https://eprint.iacr.org/undefined/undefined

[19] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," Cryptology ePrint Archive, Paper 2018/421, 2018, https://eprint.iacr.org/2018/421. [Online]. Available: https://eprint.iacr.org/2018/421

[20] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, Jan. 2020. [Online]. Available: https://doi.org/10.1007/s00145-019-09319-x

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012, https://eprint.iacr.org/2012/144. [Online]. Available: https://eprint.iacr.org/2012/144

[22] Georgieva, G. Nicolas, Mariya, C. Sergiu, Troncoso-Pastoriza, and J. Ramon, "Privacy-preserving semi-parallel logistic regression training with Fully Homomorphic Encryption," 2019, report Number: 101. [Online]. Available: https://eprint.iacr.org/2019/101

[23] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Topics in Cryptology – CT-RSA 2019*, M. Matsui, Ed. Springer International Publishing, vol. 11405, pp. 83–105, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-12612-4_5

[24] S. Halevi and V. Shoup, "Design and implementation of HElib: a homomorphic encryption library," Cryptology ePrint Archive, Paper 2020/1481, 2020, https://eprint.iacr.org/2020/1481. [Online]. Available: https://eprint.iacr.org/2020/1481

[25] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic Regression on Homomorphic Encrypted Data at Scale," *AAAI*, vol. 33, pp. 9466–9471, Jul. 2019. [Online]. Available: https://www.aaai.org/ojs/index.php/AAAI/article/view/5000

[26] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," Cryptology ePrint Archive, Paper 2019/688, 2019, https://eprint.iacr.org/2019/688. [Online]. Available: https://eprint.iacr.org/2019/688

[27] R. Inc, "Hbm2e and gddr6: Memory solutions for ai," 2020, https://go.rambus.com/hbm2e-gddr6-memory-solutions-for-ai. [Online]. Available: https://go.rambus.com/hbm2e-gddr6-memory-solutions-for-ai

[28] A. Jain, P. M. R. Rasmussen, and A. Sahai, "Threshold fully homomorphic encryption," p. 40.

[29] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 235–240. [Online]. Available: https://doi.org/10.1145/3489517.3530435

[30] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," Cryptology ePrint Archive, Paper 2021/508, 2021, https://eprint.iacr.org/2021/508. [Online]. Available: https://eprint.iacr.org/2021/508

[31] W. Jung, E. Lee, S. Kim, J. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98 772–98 789, 2021.

[32] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow, "Partial redundancy elimination in SSA form," vol. 21, no. 3, pp. 627–676. [Online]. Available: https://dl.acm.org/doi/10.1145/319301.319348

[33] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23.

New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589053

[34] J. Kim, G. Lee, S. Kim, G. Sohn, J. Kim, M. Rhu, and J. H. Ahn, "ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse." [Online]. Available: http://arxiv.org/abs/2205.00922

[35] S. Kim, J. Kim, M. J. Kim, W. Jung, M. Rhu, J. Kim, and J. H. Ahn, "BTS: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pp. 711–725. [Online]. Available: http://arxiv.org/abs/2112.15479

[36] J. Knoop, O. Rüthing, and B. Steffen, "Lazy code motion," vol. 27, no. 7, pp. 224–234. [Online]. Available: https://dl.acm.org/doi/10.1145/143103.143136

[37] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, "Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, jan 2023. [Online]. Available: https://doi.org/10.1145/3534933

[38] S. Krishnan, Z. Wan, K. Bhardwaj, P. Whatmough, A. Faust, S. Neuman, G.-Y. Wei, D. Brooks, and V. J. Reddi, "Automatic domain-specific soc design for autonomous unmanned aerial vehicles," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 300–317.

[39] S. Krishnan, A. Yazdanbakhsh, S. Prakash, J. Jabbour, I. Uchendu, S. Ghosh, B. Boroujerdian, D. Richins, D. Tripathy, A. Faust, and V. Janapa Reddi, "Archgym: An open-source gymnasium for machine learning assisted architecture design," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589049

[40] Y. Lab., "PALISADE lattice cryptography library," https://github.com/yamanalab/PALISADE, 2021.

[41] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[42] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," Cryptology ePrint Archive, Paper 2021/1688, 2021, https://eprint.iacr.org/2021/1688. [Online]. Available: https://eprint.iacr.org/2021/1688

[43] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[44] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "Performance-aware scale analysis with reserve for homomorphic encryption," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 302–317. [Online]. Available: https://doi.org/10.1145/3617232.3624870

[45] Li, S. Yongsoo, Baiyu, K. Miran, Micciancio, and Daniele, "Semi-parallel Logistic Regression for GWAS on Encrypted Data," 2019, report Number: 294. [Online]. Available: https://eprint.iacr.org/2019/294

[46] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2024.

[47] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang, J. Cong, and T. Nowatzki, "Overgen: Improving fpga usability through domain-specific overlay generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 35–56.

[48] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, p. 500–511, jun 2012. [Online]. Available: https://doi.org/10.1145/2366231.2337217

[49] C. Mouchet, J.-P. Bossuat, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: a multiparty homomorphic encryption library in go," p. 6.

[50] M. Nabeel, D. Soni, M. Ashraf, M. A. Gebremichael, H. Gamil, E. Chielle, R. Karri, M. Sanduleanu, and M. Maniatakos, "CoFHEE: A co-processor for fully homomorphic encryption execution," version: 1. [Online]. Available: http://arxiv.org/abs/2204.08742

[51] S. Narasimha, B. Jagannathan, A. Ogino, D. Jaeger, B. Greene, C. Sheraw, K. Zhao, B. Haran, U. Kwon, A. K. M. Mahalingam, B. Kannan, B. Morganfeld, J. Dechene, C. Radens, A. Tessier, A. Hassan, H. Narisetty, I. Ahsan, M. Aminpur, C. An, M. Aquilino, A. Arya, R. Augur, N. Baliga, R. Bhelkar, G. Biery, A. Blauberg, N. Borjemscaia, A. Bryant, L. Cao, V. Chauhan, M. Chen, L. Cheng, J. Choo, C. Christiansen, T. Chu, B. Cohen, R. Coleman, D. Conklin, S. Crown, A. da Silva, D. Dechene, G. Derderian, S. Deshpande, G. Dilliway, K. Donegan, M. Eller, Y. Fan, Q. Fang, A. Gassaria, R. Gauthier, S. Ghosh, G. Gifford, T. Gordon, M. Gribelyuk, G. Han, J. Han, K. Han, M. Hasan, J. Higman, J. Holt, L. Hu, L. Huang, C. Huang, T. Hung, Y. Jin, J. Johnson, S. Johnson, V. Joshi, M. Joshi, P. Justison, S. Kalaga, T. Kim, W. Kim, R. Krishnan, B. Krishnan, K. Anil, M. Kumar, J. Lee, R. Lee, J. Lemon, S. Liew, P. Lindo, M. Lingalugari, M. Lipinski, P. Liu, J. Liu, S. Lucarini, W. Ma, E. Maciejewski, S. Madisetti, A. Malinowski, J. Mehta, C. Meng, S. Mitra, C. Montgomery, H. Nayfeh, T. Nigam, G. Northrop, K. Onishi, C. Ordonio, M. Ozbek, R. Pal, S. Parihar, O. Patterson, E. Ramanathan, I. Ramirez, R. Ranjan, J. Sarad, V. Sardesai, S. Saudari, C. Schiller, B. Senapati, C. Serrau, N. Shah, T. Shen, H. Sheng, J. Shepard, Y. Shi, M. Silvestre, D. Singh, Z. Song, J. Sporre, P. Srinivasan, Z. Sun, A. Sutton, R. Sweeney, K. Tabakman, M. Tan, X. Wang, E. Woodard, G. Xu, D. Xu, T. Xuan, Y. Yan, J. Yang, K. Yeap, M. Yu, A. Zainuddin, J. Zeng, K. Zhang, M. Zhao, Y. Zhong, R. Carter, C.-H. Lin, S. Grunow, C. Child, M. Lagus, R. Fox, E. Kaste, G. Gomba, S. Samavedam, P. Agnello, and D. K. Sohn, "A 7nm cmos technology platform for mobile and high performance compute application," in *2017 IEEE International Electron Devices Meeting (IEDM)*, 2017, pp. 29.5.1–29.5.4.

[52] N. Neda, A. Ebel, B. Reynwar, and B. Reagen, "Ciflow: Dataflow analysis and optimization of key switching for homomorphic encryption," 2024. [Online]. Available: https://arxiv.org/abs/2311.01598

[53] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 416–429.

[54] S. Park, W. Song, S. Nam, H. Kim, J. Shin, and J. Lee, "Heaan.mlir: An optimizing compiler for fast ring-based homomorphic encryption," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591228

[55] M. Peter, L., "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, pp. 519–521, 1985. [Online]. Available: https://api.semanticscholar.org/CorpusID:119574413

[56] M. Poletto and V. Sarkar, "Linear scan register allocation," vol. 21, no. 5, pp. 895–913. [Online]. Available: https://dl.acm.org/doi/10.1145/330249.330250

[57] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Progress in Cryptology – LATINCRYPT 2012*, A. Hevia and G. Neven, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 139–158.

[58] V. Porpodas and M. Cintra, "CAeSaR: Unified cluster-assignment scheduling and communication reuse for clustered VLIW processors," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10.

[59] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 26–39, ISSN: 2378-203X.

[60] O. Regev, "The learning with errors problem (invited survey)," in *2010 IEEE 25th Annual Conference on Computational Complexity*, 2010, pp. 191–204.

[61] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:210836379

[62] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual*

IEEE/ACM International Symposium on Microarchitecture. ACM, pp. 238–252. [Online]. Available: https://dl.acm.org/doi/10.1145/3466752.3480070

[63] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, pp. 173–187. [Online]. Available: https://dl.acm.org/doi/10.1145/3470496.3527393

[64] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, pp. 387–398. [Online]. Available: https://ieeexplore.ieee.org/document/8675244/

[65] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 252–263, 2006.

[66] P. H. Sweany and S. J. Beaty, "Dominator-path scheduling: a global scheduling method," vol. 23, no. 1, pp. 260–263. [Online]. Available: https://dl.acm.org/doi/10.1145/144965.145824

[67] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1388–1393.

[68] O. Traub, G. Holloway, and M. D. Smith, "Quality and speed in linear-scan register allocation," vol. 33, no. 5, pp. 142–151. [Online]. Available: https://dl.acm.org/doi/10.1145/277652.277714

[69] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[70] C. Wimmer and M. Franz, "Linear scan register allocation on SSA form," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, pp. 170–179. [Online]. Available: https://dl.acm.org/doi/10.1145/1772954.1772979

[71] S.-Y. Wu, J. Liaw, C. Lin, M. Chiang, C. Yang, J. Cheng, M. Tsai, M. Liu, P. Wu, C. Chang, L. Hu, C. Lin, H. Chen, S. Chang, S. Wang, P. Tong, Y. Hsieh, K. Pan, C. Hsieh, C. Chen, C. Yao, C. Chen, T. Lee, C. Chang, H. Lin, S. Chen, J. Shieh, M. Tsai, S. Jang, K. Chen, Y. Ku, Y. See, and W. Lo, "A highly manufacturable 28nm cmos low power platform technology with fully functional 64mb sram using dual/tripe gate oxide process," in *2009 Symposium on VLSI Technology*, 2009, pp. 210–211.

[72] S.-Y. Wu, C. Y. Lin, M. C. Chiang, J. J. Liaw, J. Y. Cheng, S. H. Yang, M. Liang, T. Miyashita, C. H. Tsai, B. C. Hsu, H. Y. Chen, T. Yamamoto, S. Y. Chang, V. S. Chang, C. H. Chang, J. H. Chen, H. F. Chen, K. C. Ting, Y. K. Wu, K. H. Pan, R. F. Tsui, C. H. Yao, P. R. Chang, H. M. Lien, T. L. Lee, H. M. Lee, W. Chang, T. Chang, R. Chen, M. Yeh, C. C. Chen, Y. H. Chiu, Y. H. Chen, H. C. Huang, Y. C. Lu, C. W. Chang, M. H. Tsai, C. C. Liu, K. S. Chen, C. C. Kuo, H. T. Lin, S. M. Jang, and Y. Ku, "A 16nm finfet cmos technology for mobile soc and computing applications," in *2013 IEEE International Electron Devices Meeting*, 2013, pp. 9.1.1–9.1.4.

[73] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, C. Tsai, P. Chen, T. Miyashita, C. Chang, V. Chang, K. Pan, J. Chen, Y. Mor, K. Lai, C. Liang, H. Chen, S. Chang, C. Lin, C. Hsieh, R. Tsui, C. Yao, C. Chen, R. Chen, C. Lee, H. Lin, C. Chang, K. Chen, M. Tsai, K. Chen, Y. Ku, and S. M. Jang, "A 7nm cmos platform technology featuring 4th generation finfet transistors with a 0.027um2 high density 6-t sram cell for mobile soc applications," in *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016, pp. 2.6.1–2.6.4.

[74] Q. Xiao, S. Zheng, B. Wu, P. Xu, X. Qian, and Y. Liang, "Hasco: Towards agile hardware and software co-design for tensor computation," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1055–1068.

[75] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 870–881, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:257719794

[76] J. Zhang, X. Cheng, L. Yang, J. Hu, X. Liu, and K. Chen, "Sok: Fully homomorphic encryption accelerators," *ACM Computing Surveys*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254247068