A High-Throughput Private Inference Engine Based on 3D Stacked Memory

Zhaohui Chen[1,2,3], Ling Liang[*,1,2,3], Qi Liu[2], Zhirui Li[4,5], Fahong Zhang[2], Yanheng Lu[3,4], Zhen Gu[2,3,6]
{chenzhaohui.czh,liangling.liang,zhangyi.lq,lizhirui.lzr,zhangfahong.zfh,yanheng.lyh,guzhen.gz}@alibaba-inc.com
[1]Peking University, Beijing, China [2]DAMO Academy, Alibaba Group, Beijing, China [3]Hupan Lab, Hangzhou, China
[4]DAMO Academy, Alibaba Group, Shanghai, China [5]Xiamen University, Xiamen, China
[6]Tsinghua University, Beijing, China

ABSTRACT

Fully Homomorphic Encryption (FHE) enables unlimited computation depth, allowing privacy-enhanced neural network inference tasks directly on the ciphertext. However, existing FHE architectures suffer from the memory access bottleneck. This work proposes a High-throughput FHE engine for private inference (PI) based on 3D stacked memory (H3). H3 adopts the software-hardware co-design that dynamically adjusts the polynomial decomposition during the PI process to minimize the computation and storage overhead at a fine granularity. With 3D hybrid bonding, H3 integrates a logic die with a multi-layer embedded DRAM, routing data efficiently to the processing unit array through an efficient broadcast mechanism. H3 consumes 192mm^2 when implemented using a 28nm logic process. It achieves 1.36 million LeNet-5 or 920 ResNet-20 PI per minute, surpassing existing 7nm accelerators by 52%. This demonstrates that 3D memory is a promising technology to promote the performance of FHE.

KEYWORDS

Homomorphic encryption, accelerator, private inference

1 INTRODUCTION

Fully Homomorphic encryption (FHE) is a kind of cryptographic scheme that enables computations to be performed on encrypted data while preserving confidentiality. It offers a mathematical security guarantee, allowing users to compute in untrusted environments [1]. Modern HE algorithms have advanced to support various arithmetic operations, including addition, multiplication, and vector rotation, even on fixed-point numbers [2]. This capability has found emerging applications in private artificial intelligence inference (PI), where secure computation on sensitive data is required.

PI plays a crucial role in safeguarding privacy in domains like intelligent healthcare. During automatic medical image analysis, such as X-rays and MRI diagnosis, sensitive data can remain encrypted, ensuring it is not exposed to third parties. However, it is important to consider the trade-off between privacy and computational efficiency. In most cases, the throughput of privacy-preserving technologies, including FHE-based PI is significantly slower compared to the same semantics on plaintext. For instance, an 8-thread CPU server can complete approximately 50,000 ResNet-20 inferences on the plaintext CIFAR-10 dataset per minute. However, only 1 inference can be finished in an hour when performing the same workload with FHE-protected data [3]. The notable throughput disparity highlights the challenges when implementing privacy-preserving medical image diagnostic services.

The performance degradation of FHE primarily results from two aspects: noise control and key retention. Firstly, FHE introduces additional random noise during encryption, which is essential for security. To accommodate the noise and ensure accuracy, large parameters are necessary, which in turn complicates atomic operations. During the computation process, the noise further accumulates and consumes its budget. As the noise budget decreases during each multiplication, a resource-intensive *bootstrap* operation is required to refresh the level. Furthermore, ciphertext multiplication and vector rotation operations alter the secret key, necessitating an expensive *key-switch* operation to restore the initial key state. When the parameters are large (*i.e.* 2^{15} to 2^{17}), the key-switch keys (KSKs) and intermediate ciphertext would consume a large memory capacity.

State-of-the-art research in FHE acceleration widely acknowledges the memory-bound nature of the bootstrap process and emphasizes the importance of optimizing memory architectures. Existing effective methods to improve performance include enhancing on-chip storage and increasing memory bandwidth. To accommodate the size of intermediate data including KSKs, twiddle factors (TFs) and ciphertext, BTS [4], CraterLake [5], and ARK [6] instantiate large on-chip memory (256MB or 512MB). They also use high bandwidth memory (HBM) to minimize the data swap penalty. However, due to inherent physical constraints, the size of SRAM and the bandwidth of HBM cannot increase infinitely. Consequently, recent research has begun to explore trade-offs between precision and cost [7], and performing on-the-fly computations to reduce the need for excessive memory access by sacrificing computing [6, 8].

In this work, we developed a High throughput FHE engine using 3D stacked DRAM (H3). Current 3D memory technology provides increased bandwidth per unit capacity and lower power consumption, thereby assisting in alleviating the architecture challenges for FHE accelerators. Our contributions can be summarized as follows.

- An algorithm-hardware co-design approach that dynamically adopts ciphertext decomposition instead of storing fixed keys on-chip. We implement this approach by developing cost estimators and scheduling techniques for fine-grained optimizations.

[*] Corresponding author. This work was supported by Damo Academy through Damo Academy Research Intern Program.

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

ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3657375

- The H3 microarchitecture using the 3D stacked DRAM, which goes beyond simple bandwidth improvement. Since 3D stacked DRAM only provides vertical data ports, H3 proposes an on-chip data broadcasting mechanism, allowing each FHE logic tile to reuse shared data.
- We evaluate H3 using 28nm technology with diversified PI benchmarks. Compared to CPUs, H3 achieves a performance improvement of 3.5×10^4. Furthermore, compared to state-of-the-art 7nm accelerators, it provides 52% higher throughput with lower power consumption.

2 BACKGROUND AND MOTIVATIONS

2.1 Homomorphic Encryption

Table 1: FHE Notations.

Notation	Description
N	# polynomial coefficients
len	Length of a message, $len \leq N/2$
Q, P	Polynomial modulus, special prime modulus
L	The max multiplication level, *i.e.* initial # limbs - 1, $\prod_{i=0}^{L} q_i = Q$
l	The residual multiplication level, *i.e.* current # limbs - 1
k	# limbs of the special prime modulus, $\prod\limits_{i=l+1}^{l+k} q_i = P$
dnum	Decomposition number
α	# limbs in a group, $\alpha = \lceil (L+1)/dnum \rceil$ by default
β	$\beta = \lceil (l+1)/\alpha \rceil$
$\overline{x}$	The NTT form of a polynomial x

In this work, the CKKS scheme is referenced because of its fixed-point compatibility and wide adoption in PI applications [2]. Table 1 lists the notations we use throughout this paper. In the encryption stage, a *len*-length message vector is first encoded into a plaintext polynomial $p = \sum_{i=0}^{N-1} p_i X^i$, which is over a polynomial ring $R_Q = Z_Q[X]/\left(X^N + 1\right)$, where $p_i \in Z_Q$. Then p is scaled up by Δ and encrypted as a ciphertext (**ct**) according to $\mathbf{ct} = (a, b)$, $b = -a \cdot s + \Delta p + e$, where $a, s \in R_Q$, a is public data, and the secret key is composed of $(1, s)$, e is a small Gaussian noise that provides a secure guarantee [9]. During the decryption stage, the plaintext is recovered through $\langle \mathbf{ct}, sk \rangle = b + a \cdot s = \Delta p + e \approx \Delta p$.

HE operations. The CKKS scheme supports basic operations, including addition and multiplication between plaintext and ciphertext, as well as slot-wise rotation. Addition and multiplication are straightforwardly applied to each polynomial accordingly. To speed up polynomial multiplication , number theoretic transform (NTT) and its inverse (INTT) are appropriately applied, which reduce the complexity from $O(N^2)$ to $O(N \log N)$ [10]. We further use the residue number system (RNS)-based decomposition scheme to reduce computation costs and restrain error accumulation [11]. RNS decomposition splits a large modulus Q into $L + 1$ limbs where all q_i have the same bit-width as Δ. After multiplication, a rescale operation is required to reduce the squared scaling factor Δ^2 back to Δ by dropping one limb of the modulus, *i.e.* $l \longrightarrow l - 1$. Larger N supports a deeper multiplication level but requires higher computation costs. Rotation by r slots is implemented as a Galois automorphism operation φ, which requires a key-switch operation.

Key-switch. Whenever an operation causes the change of secret key $(1, s)$, a procedure called *key-switch* should be invoked to revert the key back to $(1, s)$. For instance, during ciphertext multiplication, the key-switch operation is necessary to revert the key from $(1, s, s^2)$ back to $(1, s)$. Similarly, during rotation, a key-switch brings the mismatched key $(1, s^{\varphi(r)})$ back to $(1, s)$.

Algorithm 1: Schoolbook key-switch

```
Function Key-switch($\overline{ksk}, \overline{ct}, l, k, \alpha, \beta, Q'_i, \hat{Q}_i, P'_i, \hat{P}_i$):
    for $i = 0 : l$ do                                    // Zero Padding & NTT → INTT
        $a_i = [\mathrm{INTT}(\overline{a}_i \cdot Q'_i)]_{q_i}$
    for $i = 0 : l + k$ do                                // ModUp & INTT → NTT
        for $d = 0 : \beta$ do
            if $d \cdot \alpha \leq i < (d+1) \cdot \alpha$ then
                $\overline{a'}_{d,i} = \mathrm{NTT}(a_i)$;
            else
                $\overline{a'}_{d,i} = \mathrm{NTT}(\sum_{j=d \cdot \alpha}^{(d+1) \cdot \alpha - 1} [a_j \cdot \hat{Q}_j]_{q_i})$;
    for $i = 0 : l + k$ do                                // Inner Product & NTT → INTT
        $a''_i = \mathrm{INTT}(\sum_{d=0}^{\beta-1} \overline{a'}_{d,i} \cdot \overline{ksk[a]}_{d,i})$;
        $b''_i = \mathrm{INTT}(\sum_{d=0}^{\beta-1} \overline{a'}_{d,i} \cdot \overline{ksk[b]}_{d,i})$;
    for $i = 0 : l$ do                                    // ModDown & INTT → NTT
        $\overline{a}_i = \mathrm{NTT}([(a''_i - \sum_{j=l+1}^{l+k-1} [a''_j \cdot \hat{P}_j]_{q_i}) \cdot P'^{-1}]_{q_i})$;
        $\overline{b}_i = \mathrm{NTT}([(b''_i - \sum_{j=l+1}^{l+k-1} [b''_j \cdot \hat{P}_j]_{q_i}) \cdot P'^{-1}]_{q_i})$;
```

The schoolbook key-switch can be summarized in Alg. 1. The inputs are the NTT form KSK $\overline{ksk}$ and the item $\overline{a}$ of NTT form ciphertext $\overline{ct}$. Specifically, $\overline{a}$ contains $l + 1$ limbs at level l. The limbs are divided into β groups, *i.e.* $Q = \prod_{i=0}^{\beta} Q_i$. Each group include α limbs, satisfying $\lceil (l+1)/\alpha \rceil = \beta$. At the beginning, $l = L$ and $\beta = dnum$, where $dnum$ is a pre-determined decomposition parameter. A key-switch procedure consists of four steps: Zero Padding, ModUp, Inner Product, and ModDown [12, 13]. Zero padding pads zero limbs to ensure all groups are composed of exactly α limbs; ModUp expands each polynomial group (α limbs) to $l + k + 1$ limbs; Inner Product performs multiplications between the $\overline{ksk}$ and the expanded ciphertext; ModDown reduces $l + k + 1$ limbs to $l + 1$ limbs. Other inputs in Alg. 1, *i.e.* k, Q'_i, $\hat{Q}_i$, P'_i, $\hat{P}_i$ are pre-computed parameters[1].

Bootstrap is a comprihensive procedure which finds $\mathbf{ct}'$ and a modulus $Q > q_0$ satisfying $\Delta p = [\langle \mathbf{ct}', sk \rangle]_Q$. The overall process consist of four steps, namely ModRaise, C2S, SinEva and S2C [12]. It starts at the initial level L and intrinsically costs L_{boot} levels, thus only $L - L_{boot}$ levels are available for effective computation.

2.2 Related Work

Existing efforts focus on reducing the overhead of key-switch. For example, Crateralake [5] implements boosted key-switch to reduce the invocation of NTT operations, while BTS [4] and ARK [6] explore optimal parameters, such as N and L, to minimize general overhead. However, these approaches assume that the KSKs are stored on-chip, which limits the parameter space due to the paradox between large key size and limited SRAM capacity.

Another challenge in FHE implementation is the bootstrap operation which dominates the overall latency. As we know, bootstrap is memory-bound, for which HBM can help. Nevertheless, with logic density rapidly increasing, more techniques like on-the-fly limb generation and key regeneration would be necessary [5, 6]. Unfortunately, these techniques come with increased computational demands.

[1] These parameters are derived from Q_i and P_i, $Q_i = \prod_{j=d\alpha}^{(d+1)\alpha-1} q_j$, $Q'_i = \frac{Q}{Q_i} \cdot \prod_{j=l+1}^{l+k-1} q_j$; $\hat{Q}_i = [\frac{q_i}{Q_i}]_{q_i} \cdot \frac{Q_i}{q_i}$; $P' = P \cdot \prod_{j=l+1}^{l+k-1} q_j$; and $\hat{P}_i = [\frac{q_i}{P}]_{q_i} \cdot \frac{P}{q_i}$.

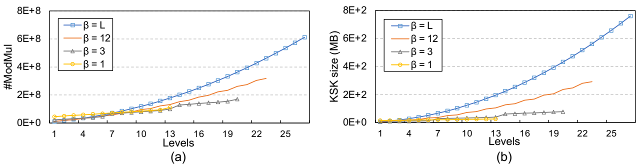


Figure 1: Comparison of key-switch complexity under different parameter settings: (a) #ModMul and (b) KSK size for one key-switch under different l and β.

2.3 Motivations for using 3D DRAM in PI

PI services have two essential requirements: high throughput for optimal performance and low power consumption to minimize economic costs. To meet these needs, the current industry-leading solution is to integrate the 3D embedded DRAM. This approach involves stacking multiple layers of LPDDR4/4X dies using hybrid bonding (HB) and mini-TSV technology. Unlike HBM, 3D DRAM eliminates the necessity for cross-interposer wire connections and offers smaller bump sizes with higher interface pin densities. Consequently, 3D DRAM offers higher bandwidth per unit capacity and lower power consumption compared to HBM.

3 OPTIMIZATING FHE WORKFLOW

In this section, we begin by presenting a dynamic key-switch parameter selection algorithm achieving the lowest cost. Furthermore, we introduce an inner key-switch scheduling technique to overlap the latency. Eventually, we develop an instruction set architecture (ISA) that facilitates the software-hardware co-design.

3.1 Dynamic Key-Switch Adaptation

In FHE applications, the operation complicity and supported levels vary significantly according to parameter settings. When N is fixed, the complexity of key-switch under different levels l varies with decomposition, *i.e.* the value of α, β. As shown in Fig. 1, when $N = 2^{16}, l = 20$, the computation complexity and KSK size for $\beta = L$ is 2.4× and 5.5× larger than the case $\beta = 3$. However, smaller β cannot support deeper multiplication depth l. Therefore, it is crucial to choose an appropriate α in a given FHE context.

Most previous studies adopt fixed parameters for different multiplication level l [4, 6, 14, 15]. In contrast, we dynamically adapt the α for different levels l. To achieve dynamic adaption, we first build a cost estimator on both computation and memory access. Since the major workloads are coefficient-wise multiplication (CWM) and NTT, we estimate the overhead of CWM as

$$\#\text{CWM} = \underbrace{(l+1)}_{\text{Zero Padding}} + \underbrace{(\beta\cdot(l+k+1)-(l+1))\cdot\alpha}_{\text{ModUp}} + \underbrace{2\cdot\beta\cdot(l+k+1)}_{\text{Inner Product}} + \underbrace{2\cdot(l+1)\cdot(k+1)}_{\text{ModDown}}, \quad (1)$$

and the amount of NTTs/INTTs equals

$$\#\text{(I)NTT} = \underbrace{(l+1)}_{\text{after Zero Padding}} + \underbrace{\beta\cdot(l+k+1)}_{\text{after ModUp}} + \underbrace{2\cdot(l+k+1)}_{\text{after Inner Product}} + \underbrace{2\cdot(l+1)}_{\text{after ModDown}}. \quad (2)$$

On estimating the memory access overhead, we gather the total capacity of each key-switch and plaintext during bootstrap.

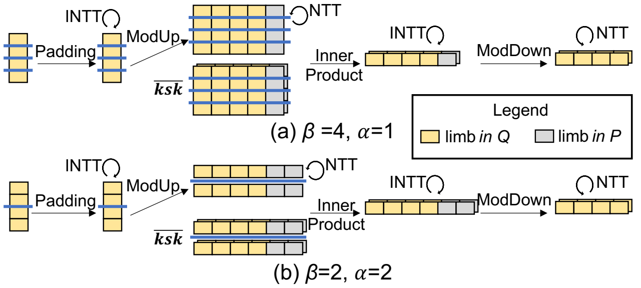


Figure 2: An example of the optimized key-switch, in which the modulus $Q \cdot P$ can accommodate a maximum of 6 limbs.

Algorithm 2: Dynamic key-switch

```
Function Dynamic_key-switch ($l$):
    $cost = MAX$;
    $l', k', \alpha', \beta' = NULL$;
    for $\alpha = 1$; $\alpha \leq l+1$; $\alpha$++ do
        $\beta = \lceil (l+1)/\alpha \rceil$;
        $k = \alpha$;
        // Estimate multiplication based on Eq. 1 & 2
        $cost\_est$ =CostEstimation$(l, k, \alpha, \beta)$;
        if $cost\_est < cost$ then
            $l', k', \alpha', \beta' = l, k, \alpha, \beta$;
    $\overline{ct}$ = Key-switch($\overline{evk}, \overline{ct}, l', k', \alpha', \beta', Q'_i, \hat{Q}_i, P'_i, \hat{P}_i$)
```

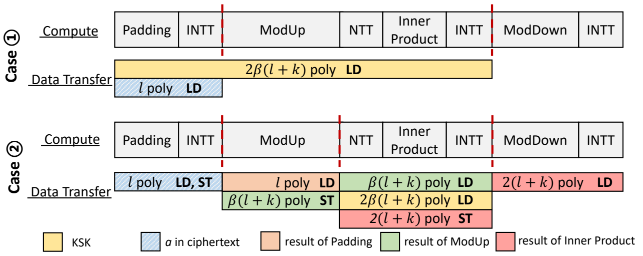


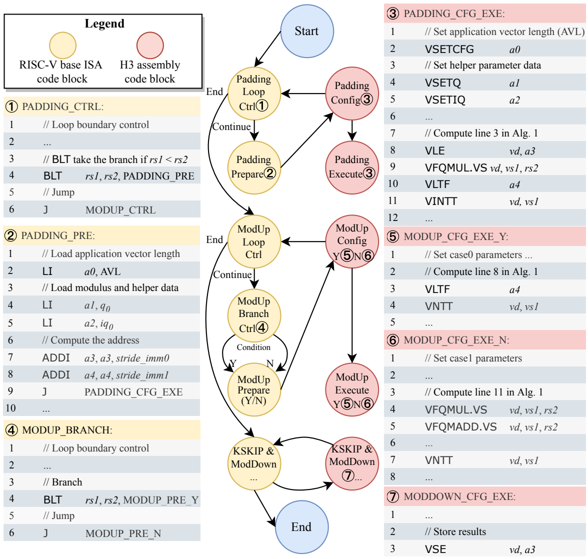
Figure 3: Two execution flows of key-switch, where block length does not reflect execution time or data size.

The detailed methodology of parameter selection is shown in Alg. 2. In each level l, we traverse all possible values of α and use the cost model in Eq. 1 & 2 to determine the best parameter setting. Note that the NTT after ModUp always occupies most of the computation cost as Eq. 2, which is also proportional to the KSK size. Thus, a smaller computation complexity has less memory cost in most scenarios. An example is presented in Fig. 2. The optimized algorithm selects the parameter set in Fig. 2(b), which reduces #(I)NTT by 6 and the $\overline{ksk}$ size by 40%. The detailed benefits of our optimized algorithm are presented in Sec. 5.3.

3.2 Fine-Grained Key-Switch Scheduling

Since the complexity of key-switch varies a lot under different parameter settings, a dedicated execution flow design is required. Based on the SRAM capacity, we designed two execution flows to handle different key-switch settings as Fig. 3.

Case①. In the first case, we assume that the SRAM is capable of storing the input of key-switch and the result before ModDown ($3l+2k$ polynomials). In this case, we only need to load the $\overline{ksk}$ from off-chip memory, which overlaps the operations before ModDown.



```
① PADDING_CTRL:
    // Loop boundary control
    ...
    // BLT take the branch if rs1 < rs2
    BLT       rs1, rs2, PADDING_PRE
    // Jump
    J         MODUP_CTRL
```

```
② PADDING_PRE:
    // Load application vector length
    LI        a0, AVL
    // Load modulus and helper data
    LI        a1, q0
    LI        a2, iq0
    // Compute the address
    ADDI      a3, a3, stride_imm0
    ADDI      a4, a4, stride_imm1
    J         PADDING_CFG_EXE
    ...
```

```
④ MODUP_BRANCH:
    // Loop boundary control
    ...
    // Branch
    BLT       rs1, rs2, MODUP_PRE_Y
    // Jump
    J         MODUP_PRE_N
```

```
③ PADDING_CFG_EXE:
    // Set application vector length (AVL)
    VSETCFG        a0
    // Set helper parameter data
    VSETQ          a1
    VSETIQ         a2
    ...
    // Compute line 3 in Alg. 1
    VLE            vd, a3
    VFQMUL.VS vd, vs1, rs2
    VLTF           a4
    VINTT          vd, vs1
    ...
```

```
⑤ MODUP_CFG_EXE_Y:
    // Set case0 parameters ...
    // Compute line 8 in Alg. 1
    VLTF           a4
    VNTT           vd, vs1
    ...
```

```
⑥ MODUP_CFG_EXE_N:
    // Set case1 parameters
    ...
    // Compute line 11 in Alg. 1
    VFQMUL.VS      vd, vs1, rs2
    VFQMADD.VS     vd, vs1, rs2
    ...
    VNTT           vd, vs1
    ...
```

```
⑦ MODDOWN_CFG_EXE:
    ...
    // Store results
    VSE            vd, a3
```

Figure 4: Workflow and example assembly of H3 when executing key-switch as a RISC-V coupled instruction processor.

Case②. The second execution flow mainly tackles the situation that the SRAM is insufficient to store all intermediate results. In this case, each operator in key-switch must load and store the intermediate data, which requires frequent data transfer between SRAM and off-chip memory.

3.3 Software Hardware Co-Design

Table 2: The H3 ISA Definition.

Instruction	Description
VSET<PARAM>	Set FHE parameters
VFQ<ADD,MUL.SHIFT>	Coefficient-wise operation over Z_{q_i}
VNTT/VINTT	Vector NTT and INTT
VLE/VSE	Vector load and store

To implement the desired functionality on hardware, we define custom 32-bit R-type instructions compatible with the RISC-V architecture. These custom instructions are outlined in Table 2. Fig. 4 illustrates the program execution process of key-switch as a RISC-V coupled processor. The RISC-V standard instruction blocks control the loops and jumps as code block ①. Block ② undertakes simple logic such as fetching scalar values and address increments. The instructions ③ first configure the vector length, *etc.*, allowing subsequent instructions to execute FHE arithmetic instructions according to application demands. When standard instructions and H3 executes on separate processor backends, they can run asynchronously. Data hazard is statically analyzed by the compiler, which inserts synchronization to maintain memory consistency. Code blocks ④, ⑤, and ⑥ demonstrate how the program supports branching through RISC-V standard instructions. Once the program ends, the ciphertext data can be written back to memory, as shown in code block ⑦. Throughout the execution flow, the ciphertext computation is always accelerated by H3 processor, which demands an FHE-specific microarchitecture.

4 3D MICROARCHITECTURE DESIGN

4.1 Design Overview

The H3 system architecture incorporates a 4-layer 3D DRAM stacked with a dedicated FHE-specific logic die at the bottom layer, as shown in Fig. 5(a). In the integration process, mini-TSVs are utilized for vertical connectivity, requiring the DRAM tile and logic tile to be aligned in the vertical direction. Each logic tile can access the aligned DRAM for local data, and the neighbors for shared data as Fig. 5(b). The DRAM die is divided into 4 × 4 tiles, offering 135 MBps/Gb. Each DRAM tile has a capacity of 768Mb as Fig. 5(c), resulting in an overall capacity of 6GB.

The FHE logic die is comprised of 4 × 4 isomorphic tiles, each containing a lightweight RISC-V core [16], an FHE instruction coprocessor unit (FHEU), SRAM-based architectural registers, and a NoC router, as depicted in Fig. 5(d). The FHEU consists of an array of execution lanes, each equipped with a modular arithmetic unit (MAU) capable of executing element-wise instructions in a single instruction multiple data (SIMD) fashion. By default, the FHEU consists of 512 lanes running at 1GHz. In this configuration, H3 achieves a peak performance of 8T MAC per second. Additionally, the NoC can support a read-and-write throughput of 512 words per channel port.

4.2 Data Broadcast Based on Embedded DRAM

H3 employs a multi-core synchronous execution approach for efficient data sharing. The common data is initially stored interleaved within the DRAM tiles. During the execution phase, the tiles first access the vertically aligned banks. Subsequently, they conduct all-to-all data broadcasts on the NoC. H3 implements a mesh network instead of a torus due to the difficulties in fabricating additional physical connections between the edge tiles. As depicted in Fig. 6, the 4 × 4 nodes are divided into three categories based on their topological properties: A (corners), B (margins), and C (center). Class A, *i.e.* the corner nodes, pose a bottleneck among these classes as they have fewer available data channels. To address the challenge of broadcasting common data among the logic tiles, H3 executes the following routing algorithm to complete data broadcast within 8 hops. Fig. 6 takes the process of broadcasting to A_1 as an example.

Hop 1~3 (Fig. 6(a)). In the row direction, each node sends its data to the nodes in the same row hop-by-hop. For example, node A_1 receives data from nodes B_1, B_0, and A_0 in sequence. Similarly, in the column direction, node A_1 receives data from nodes B_3, B_5, and A_3 in sequence. After completing these three hops, each node n_{ij} aggregates all the data blocks of its row $r_i = n_{ij}, j \in \{0, 1, 2, 3\}$ and column $c_j = n_{ij}, i \in \{0, 1, 2, 3\}$.

Hop 4~6 (Fig. 6(b)). In each row, the four nodes broadcast the data block received from the farthest row in Hop 1~3. More specifically, $r_0 \sim r_3$ transmit the data originally load from r_3, r_3, r_0, and r_0. In the column direction, each node receives data from its adjacent nodes corresponding to the adjacent rows. Continuing with the example, $r_0 \sim r_3$ will respectively receive data blocks from r_1, r_2, r_1, and r_2. After completing this stage, each node n_{ij} only misses at most 3 data blocks $r_{\notin} \setminus n_{\notin j}$, where the $r_{\notin}$ is the missing row. Specifically $r_0 \sim r_3$ miss r_2, r_0, r_3, r_1 data, respectively.

Hop 7~8 (Fig. 6(c)). To aggregate the last incomplete row, each node can obtain the complete $r_{\notin}$ from the neighboring nodes simultaneously in the row and column directions. In Fig. 6(c), A_1 aggregates data blocks of C_3, C_2, and B_4. Due to the conflict-free nature of the above routing algorithm, the broadcast process can be executed in a pipeline fashion.

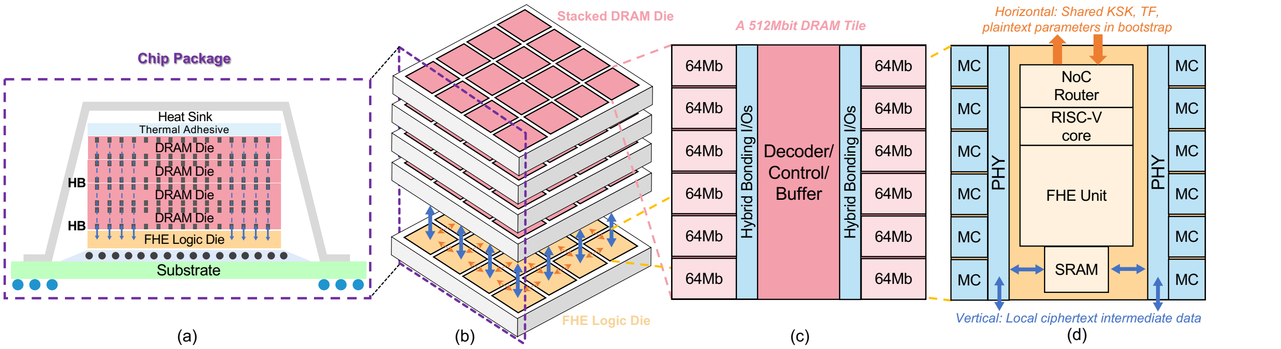


Figure 5: An overview of H3 microarchitecture: (a) the vertical cross-section, (b) the 3D integration, (c) a DRAM tile and (d) a FHE-specific tile.

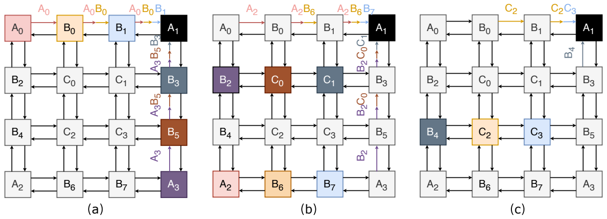


Figure 6: Datapath of routing all data to node n_{03} in 8 hops across the 4 × 4 tiles, where A_1 is a topological alias n_{03}. All 16 nodes gather the complete 16 data blocks during the broadcast process.

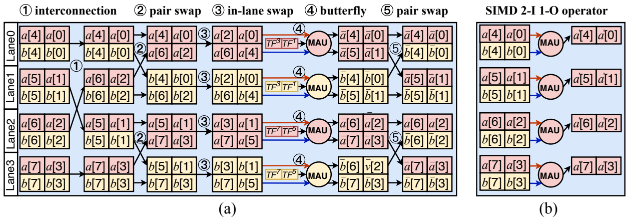


Figure 7: Datapath of (a) parallel executing $\bar{a}$ = NTT(a), $\bar{b}$ = NTT(b), and (b) a' = CWM(a, b) with 4 parallel MAUs.

4.3 Flexible Arithmetic Dataflow

H3 facilitates flexible dataflow to execute NTT and coefficient-wise arithmetic in the FHEU, which is a unified MAU array. We adopt the Barrett reduction algorithm [17] and support configurable prime modulus under 36 bits [7].

The NTT datapath (Fig. 7 (a)). An N-length NTT has $\log(N)$ stages with $N/2$ butterfly in each stage [18, 19]. Fig. 7(a) illustrates the datapath of two parallel 8-point NTT in 4-lane settings, *i.e.* $\bar{a}$ = NTT(a), $\bar{b}$ = NTT(b). In this case, the two 8-point vectors, a and b, are stored separately in four lanes within two continuous addresses. In each lane, the coefficients are stored in two physical banks marked in different colors. The execution process of a stage consists of 5 steps. In step ①, the elements are shuffled through the lane interconnection path. In ②, two adjacent lanes exchange data within the pair. Step ③ exchanges data between the two pipeline stages through buffer and bypass logic. In step ④, the butterfly is performed using a general MAU. In ⑤, the elements are swapped again between the lane pairs to restore the initial data arrangement.

Coefficient-wise datapath (Fig. 7 (b)). The coefficient-wise operation in our architecture is executed in SIMD fashion. This is made possible by storing the polynomials in a parity isolation distribution. As a result, two operands can simultaneously feed an MAU, and the result can be directly written into the destination.

Table 3: Area and power.

Component	Area (mm^2)	Power (W)
FHEU logic	7.05	3.13
SRAM	1.87	0.75
RV core & L1$	0.51	0.23
MC/PHY/Misc.	2.57	0.35
Total FHE tile	12.00	4.46
FHE logic die	192.00	71.36
Stacked DRAM	192.00	14.40

Table 4: Comparision on accelerators.

Engine*	Tech.	Area (mm^2)	Power (W)	BW. (GB/s)
H3	28nm	192.0	85.8	6480
F1 [8]	12/14nm	151.4	180.4	1024
BTS [4]	7nm	373.6	163.2	2048
CLake [5]	12/14nm	472.3	~ 320.0	1024
ARK [6]	7nm	418.3	281.3	1024
SHARP [7]	7nm	178.8	178.8	1024

* The area and power consumption of off-chip memory are not included except for H3.

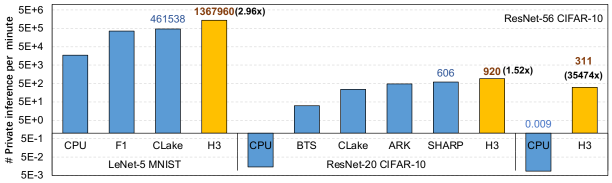


Figure 8: Throughput comparison on LeNet-5 and ResNet PI.

5 EVALUATION

This section evaluates the H3 architecture, which provides a substantial performance boost in PI applications. Moreover, we analyze the effect of critical architectural settings on performance.

5.1 Area and Power

We implemented the FHE logic die in RTL and evaluated it using the Synopsys Design Compiler with a 28nm library. The 4-layer stacked DRAM is assessed using recent releases [20, 21]. To maintain vertical alignment, each die is split into 16 isomorphic tiles. The component level area and power consumption are provided in Table 3. Notably, H3 does not require an advanced process node (5nm/7nm), resulting in significantly lower manufacturing costs compared to other alternatives.

5.2 Performance on PI Benchmarks

We develop an instruction-level analytical model that supports dependency analysis and memory management for the following assessments.

LeNet-5 (CryptoNets-LoLa [22]) is a 5-layer network that replaces the activation layers with square functions. The evaluation is conducted on encrypted input and unencrypted model weights with the MNIST dataset. H3 achieves a throughput of 1.36 million per minute, which is 2.96 × higher than that of CraterLake [5].

ResNet-20 and ResNet-56 [3] are high-accuracy deep learning networks. The PI process involves multiple bootstrap operations,

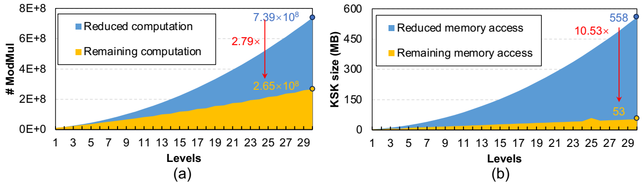


Figure 9: Comparisons of (a) computation and (b) KSK memory access overhead between the schoolbook and the proposed approach, in which $N = 2^{16}$ with 14 bootstrap levels and 16 effective levels.

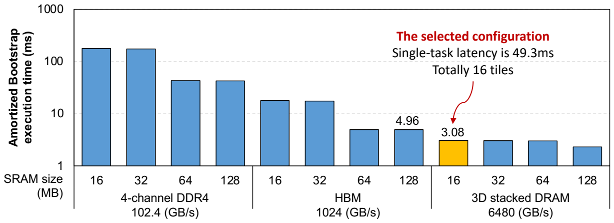


Figure 10: Performance analysis on the bootstrap under different SRAM sizes and device memory bandwidth settings.

accounting for 85.07% of the overall latency cost. However, a CPU can only achieve 1.56 ResNet-20 PI/hour, which is 1.9×10^6 lower than plaintext. Fortunately, H3 improves the performance by approximately 35,000×, achieving 920 PI/min, which is 52% better than SHARP [7]. To further improve accuracy, H3 provides the capability to process 311 ResNet-56 PI/min.

5.3 Sensitivity Studies

Analysis on dynamic key-switch adaption. Our proposed algorithm decreases the key-switch computation overhead on H3 architecture throughout multiplication levels 1 ∼ 30. On the storage problem, storing each KSK on-chip requires 558/2=229MB after utilizing the polynomial a regenerating technique [5, 6]. In contrast, H3 loads the KSKs on the fly from the high-bandwidth 3D memory. By utilizing Alg. 2, only 53MB is required at the max level.

Analysis on memory architecture. We use the performance of bootstrap, which is the major workload in ResNet, to identify the most appropriate configuration for the memory bandwidth and SRAM size. The results are shown in Fig. 10. The performance of architectures based on 4-channel DDR4 and HBM is evaluated with the same peak performance. The HBM-based architecture has a lower single-task latency which is 4.96ms, whereas the synchronous execution of H3 tiles achieves a better-amortized cost, which improves the throughput by 61%.

On the SRAM capacity settings, performing the key-switch operation in Case② (see Sec. 3.2) involves significant data exchange with the device memory. To handle this efficiently, a 64MB SRAM can accommodate Case① in the worst-case scenario (with $N = 2^{16}$ and the maximum number of limbs), allowing the intermediate data of a single key-switch operation to reside on-chip. This brings notable performance benefits in DDR4 and HBM settings. In the 3D memory dataflow, the SRAM is distributed among the 16 logic tiles. Storing all 16 intermediate ciphertexts on-chip simultaneously is not feasible. Nevertheless, the data exchange between on/off-chip costs less than the counterparts. Considering both resource consumption and execution latency, instantiating 1MB SRAM per tile is the optimal choice. To elaborate, increasing the SRAM capacity to 64MB only provides 2% performance improvement.

6 CONCLUSION

Privacy-preserving inference, such as X-ray and MRI analysis, presents challenges in terms of throughput. However, traditional memory architectures can no longer achieve further performance improvements. This paper introduces H3, which enables high-throughput FHE inference with a dynamic key-switch algorithm that minimizes computation and memory overhead using a fine-grained cost model. Furthermore, we develop a 3D stacked architecture with carefully designed data broadcast and efficient acceleration ISA. H3 achieves a throughput of 1.36 million LeNet-5 or 920 ResNet-20 PI with 28nm, outpacing existing 7nm accelerators by 52%. This demonstrates that 3D memory is a highly promising technology that can resolve performance limitations in FHE.

REFERENCES

[1] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. of STOC 2009*, pages 169–178. ACM, 2009.

[2] Jung Hee Cheon et al. Homomorphic encryption for arithmetic of approximate numbers. In *Proc. of ASIACRYPT 2017*, pages 409–437. Springer, 2017.

[3] Eunsang Lee et al. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *Proc. of ICML 2022*, pages 12403–12422. PMLR, 2022.

[4] Jongmin Kim et al. ARK: fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *Proc. of MICRO 2022*, pages 1237–1254. IEEE, 2022.

[5] Nikola Samardzic et al. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proc. of ISCA 2022*, pages 173–187. ACM, 2022.

[6] Sangpyo Kim et al. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *Proc. of ISCA 2022*, pages 711–725. ACM, 2022.

[7] Jongmin Kim et al. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proc. of ISCA 2023*, pages 18:1–18:15. ACM, 2023.

[8] Nikola Samardzic et al. F1: A fast and programmable accelerator for fully homomorphic encryption. In *Proc. of MICRO 2021*, pages 238–252. ACM, 2021.

[9] Albrecht et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021.

[10] Patrick Longa et al. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *Proc. of CANS 2016*, pages 124–139, 2016.

[11] Jung Hee Cheon et al. A full RNS variant of approximate homomorphic encryption. In *Proc. of SAC 2018*, pages 347–368. Springer, 2018.

[12] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In *Proc. of CT-RSA 2020*, pages 364–390. Springer, 2020.

[13] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In *Proc. of ASIACRYPT 2021*, pages 608–639. Springer, 2021.

[14] Robin Geelen et al. BASALISC: flexible asynchronous hardware accelerator for fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.

[15] Wonkyung Jung et al. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):114–148, 2021.

[16] Ltd T-Head Semiconductor Co. T-head-semi/opene906, Aug 2022. https://github.com/T-Head-Semi/opene906.

[17] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Proc. of CRYPTO 1986*, pages 311–323. Springer, 1986.

[18] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[19] Xuanle Ren et al. CHAM: A customized homomorphic encryption accelerator for fast matrix-vector product. In *Proc. of DAC 2023*, pages 1–6. IEEE, 2023.

[20] Dimin Niu et al. 184QPS/W 64Mb/mm^{2}3D logic-to-DRAM hybrid bonding with process-near-memory engine for recommendation system. In *Proc. of ISSCC 2022*, pages 1–3. IEEE, 2022.

[21] Song Wang et al. A 135 GBps/Gbit 0.66 pJ/bit stacked embedded DRAM with multilayer arrays by fine pitch hybrid bonding and mini-TSV. In *Proc. of VLSI 2023*, pages 1–2. IEEE, 2023.

[22] Alon Brutzkus et al. Low latency privacy preserving inference. In *Proc. of ICML 2019*, pages 812–821. PMLR, 2019.