2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00090

Hydra: Scale-out FHE Accelerator Architecture for Secure Deep Learning on FPGA

Yinghao Yang*†, Xicheng Xu*†, Haibin Zhang, Jie Song, Xin Tang, Hang Lu*†‡, Xiaowei Li*†‡
* State Key Laboratory of Processors, Institute of Computing Technology, CAS, Beijing, China.
† University of Chinese Academy of Sciences, Beijing, China.
‡ Zhongguancun Laboratory, Beijing, China.
{yangyinghao21b, luhang, lxw}@ict.ac.cn, xuxicheng20@mails.ucas.ac.cn,
zhangli14180206@163.com, songjie_email@126.com, tangxin85@126.com

***Abstract*—Deep learning, including Convolutional Neural Network (CNN) and Large Language Model (LLM), under Fully Homomorphic Encryption (FHE) is very computationally intensive because of the burdensome computations like ciphertext convolution and matrix multiplication, non-linear layers, and bootstrapping. Existing FHE accelerators focus on the high throughput computational units, stacking parallelized clusters to maximize ciphertext inference performance. Nevertheless, this design philosophy cannot leverage the substantial parallelism at the application level and is not scalable for further performance enhancement by simply adding additional compute nodes to cope with the ever-increasing model sizes in the future. In this paper, we propose the high-performance FHE acceleration architecture in a "scale-out" manner for secure deep learning, termed as *Hydra*. It supports the multi-server scaling and arbitrary computational nodes theoretically, each handling a portion of the deep learning model governed by the central scheduling mechanism on the host server. Hydra exhibits excellent scalability and delivers outstanding performance across a range of compute resource sizes. We highlight the following results: (1) up to 74× and 160× speedup over the SOTA single card accelerator Poseidon and FAB; (2) outperforms 8-card FAB-2 by 12× to 21× for FHE-based CNNs and LLMs; (3) outperforms SOTA ASIC accelerators, CraterLake and SHARP, by 8.1× and 2.5× for LLM OPT-6.7B, and achieves comparable or superior energy efficiency under the same chip technology.**

I. Introduction

Data asset, vital for production, plays a significant role in various aspects of social development. However, the sensitivity and privacy of much of this data make it challenging to circulate efficiently and fully realize its value. This has led to an urgent need for privacy computing technologies that protect data from leakage while enabling its utilization. Fully Homomorphic Encryption (FHE hereafter), as a mainstream technology for privacy computing, supports computing on encrypted data (i.e., data remains usable but invisible). FHE caters to many data privacy protection needs [1]–[6]. Deep learning (DL), including Convolutional Neural Network (CNN) and Large Language Model (LLM), as widely researched and used methods of artificial intelligence, also face the problem of data leakage. Efficient FHE-based CNN and LLM inference methods have been proposed to protect client privacy in cloud datacenter scenarios [7]–[13]. The industry is also focusing on the deep learning applications using FHE. For example, Amazon has enhanced its SageMaker endpoints using FHE [14] for clients to perform real-time and secure inference.

However, FHE-based deep learning tasks are inefficient due to the large expanded ciphertext size and complex calculations, being 4 to 5 orders of magnitude slower than plaintext. Many previous FHE-specific hardware accelerators, including ASIC, and FPGA designs [15]–[29], pursue the high-throughput computational units (CUs) by optimizing data flow and scaling the hardware resources, i.e., memory capacity and bandwidth, and number of CUs. Existing accelerator architectures aim to improve the efficiency of low-level FHE operators by tapping its parallelism without paying attention to the characteristics of the FHE application itself. There is significant application-level parallelism in FHE-based DL inference, e.g., inter-kernel parallelism in the convolutional layer of CNN and matrix multiplications in LLM. These parallelisms are beyond the utilization capacity of traditional accelerators which only focus on the basic operators, such as Number Theoretic Transform (NTT), Modular Addition (MA), Modular Multiplication (MM), etc. Therefore, compared to the traditional FHE accelerator, scale-out accelerator architecture has a greater affinity for DL inference. The scale-out architecture involves multiple, scalable, and collaborated computing nodes, that can take advantage of both the low-level operator of FHE and the application-level parallelism of DL models, and cope well with the computational overhead inflation associated with increasing model scale. Although it introduces issues of task allocation and multi-node state synchronization, this architecture can transcend the limitations of a single computing node, expand by adding nodes, and freely augment resources when dealing with large-scale secure DL models.

In this paper, we propose a high-performance scale-out accelerator architecture that supports multi-server scaling, Hydra,

Corresponding authors are Yinghao Yang and Hang Lu. This work was supported in part by the National Natural Science Foundation of China under Grant 62172387; in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences (CAS) under Grant 2021098; in part by the CCF-Phytium Fund 2023; in part by the Open Research Fund of the State Key Laboratory of Blockchain and Data Security, Zhejiang University.

TABLE I
PARALLELISM OF FHE-BASED DL INFERENCE. WE LIST THE MAX. AND MIN. NUMBER OF PARALLELISM AND ACTIVATION CIPHERTEXTS IN DIFFERENT LAYERS OF FOUR FHE-BASED DL MODEL INFERENCE IMPLEMENTATIONS, WHERE THE MODELS ARE IMPLEMENTED USING THE SOTA CNN AND LLM APPROACHES PROPOSED IN [12], [13], [30] WITH THE SAME PARAMETER SETTINGS AS SHARP [21], I.E., LOG(PQ)=1692 AND LOGQ=1260. RESNET-18 AND RESNET-50 USE IMAGENET DATASET WITH THE INPUT SIZE OF 224×224. OPT-6.7B HAS A SIMILAR ARCHITECTURE TO BERT-BASE, WE IMPLEMENT IT FOLLOWING THE METHOD IN [13]. BERT-BASE AND OPT-6.7B USE THE INPUT SEQUENCE SIZE OF 128×768 AND 200×4096.

Model / Layer	ResNet-18 (Min./Max.)	ResNet-50 (Min./Max.)	BERT-base (Min./Max.)	OPT-6.7B (Min./Max.)	Rotation	CMult	PMult	HAdd
ConvBN	384 / 1,024	384 / 1,024	NA	NA	8	0	2	7
Pooling	6 / 64	12 / 256	NA	NA	2	0	1	0
FC	1,511 / 1,511	3,047 / 3,047	NA	NA	1	0	1	0
PCMM	NA	NA	98,304 / 393,216	153,600 / 614,400	1	0	1	0
CCMM	NA	NA	384 / 384	1000 / 1000	7	1	1	6
Non-linear	4 / 128	4 / 128	4 / 48	8 / 72	0	8	0	15
Ciphertext	1 / 32	1 / 32	1 / 12	2 / 18	-	-	-	-

for FHE-based DL inference. Rather than solely focusing on optimizing the basic computation unit of FHE, Hydra aims to explore the optimization of computational task allocation and communication for FHE-based DL inference in a scale-out architecture, including the design of efficient task mapping and communication strategy, and synchronization mechanism. The contributions of this paper are listed as follows:

- *We propose the practical high-performace scale-out accelerator architecture for secure DL inference – Hydra.* It targets FHE-based CNN and LLM inference and offers excellent scalability supporting multi-server scaling.
- *We propose an efficient task mapping strategy and synchronization mechanism for the scale-out architecture.* We design specialized task mapping methods for each key procedure in DL inference including bootstrapping. By coordinating computational and communication, Hydra can achieve considerable performance.
- *We implement three prototypes of Hydra, i.e., Hydra-S, Hydra-M, and Hydra-L, based on the commercial FPGA platform, Xilinx Alveo U280, to evaluate its real-world performance.* We highlight the following results: (1) up to 88× to 160× speedup over the SOTA FPGA design FAB in FHE-based LLM; (2) under the same number of cards, Hydra achieves a 2.8× to 3.3× performance improvement compared to the 8-card architecture of FAB-2; (3) communication overhead in Hydra-M and Hydra-L is only 0.04% and 1.4% respectively on OPT-6.7B.

II. WHY SCALE-OUT?

The intractable computational complexity and intensive memory access make FHE-based DL inference challenging to deploy in practice. For the ResNet-20 for CIFAR-10, a tailored small-scale model, the most advanced practical accelerators, i.e., Poseidon [19] and FAB [18], achieve a performance of nearly 3 seconds. ASIC designs can achieve better performance than FPGAs by stacking large amounts of resources, but these efforts are still at the simulation stage and do not support scaling for further performance improvements. Although FAB proposes a scale-out acceleration architecture, it is only a relatively simple point-to-point interconnection of FPGA computing nodes to accelerate computation on different ciphertexts in parallel, and there is no special task allocation and efficient communication and synchronization mechanism for DL, making it difficult to efficiently support DL applications under FHE. Furthermore, Liberate-FHE [31] is an excellent open-source library supporting multi-GPU FHE acceleration. Its unique engine class allows users to specify the GPU card allocation and flexible data object transfer between cards when creating ciphertext or key objects, providing great programming flexibility. However, Liberate-FHE does not provide a multi-GPU task allocation strategy for FHE-based DL inference, which would help users better utilize it for building multi-GPU-accelerated DL applications. GPU-based [32] supports multi-card scaling, but only performs simple multi-card task partitioning for very small CNN models without bootstrapping, making it unsuitable for real DL models.

A. The Affinity to DL Parallelism

FHE-based DL inference is commonly implemented based on the CKKS scheme with a large polynomial degree [12], [13], [30], and the FHE operations have large overheads, such as Homomorphic Addition (HAdd), Ciphertext-Ciphertext Multiplication (CMult), Plaintext-Ciphertext multiplication (PMult), Rescale, Keyswitch, Rotation, and Bootstrapping, are frequently utilized. Every operation is performed under the large polynomial and comprises several basic operators, such as NTT, MA, MM, etc. Limited by resources and physical limits, accelerators both based on FPGA and ASIC only support maximum parallelism of 512 and 2048 [15], [16], [18], [19], [33], far from that in FHE operators.

Fortunately, in addition to the basic operators, parallelism also exists at the application level in secure DL inference. Table I lists the application-level parallelism in four CNN and LLM implementations under FHE proposed in [12], [13], [30], i.e., ResNet-18, ResNet-50, BERT-base, and OPT-6.7B. The key computations in CNNs include ConvBN (Convolution and Batch Normalization), Non-linear, Pooling, FC (Fully Connection), and Bootstrapping. In LLMs, the key computations include PCMM (Plaintext-Ciphertext Matrix Multiplication), CCMM (Ciphertext-Ciphertext Matrix Multiplication), and the same, Non-linear, and Bootstrapping as in CNNs. ResNet-18 and ResNet-50 exhibit a ConvBN parallelism ranging from 384 to a staggering 16384. Each parallel computing unit encapsulates multiple FHE operations, i.e., 8 Rotations, 2 PMults, and 7 HAdds. Both pooling and FC exhibit a considerable degree of parallelism. Pooling, essentially a convolution calculation with a single input channel, shares its parallelism source with ConvBN, FC, fundamentally a matrix-vector multiplication, derives its parallelism from the weight matrix size. ResNet-18 and ResNet-50, with their substantial FC parameters, introduce significant parallel computations. The unique PCMM and CCMM operations in LLM are richer in parallelism, up to tens of thousands, due to the matrix size is generally large and the data operations between rows and columns are independent. Based on the implementation in [13], the PCMM parallel unit contains 1 rotation and 1 PMult operation, while the CCMM's contains 7 rotations, 1 CMult, 1 PMult, and 6 HAdd. The parallelism of Non-linear emanates from its multiple parallel polynomial evaluations, with all ciphertexts storing activation values in the layer necessitating identical computations, thereby offering substantial parallelism with a maximum of 128 in CNNs and 72 in LLMs. Lastly, we employ the required number of bootstraps as the parallelism metric for bootstrapping, contingent on the number of ciphertexts in the current layer. CNNs and LLMs exhibit a maximum bootstrapping parallelism of 32 and 18, indicating the need to compute multiple bootstraps simultaneously in DL models. As the most intricate FHE operation, bootstrapping also harbors internal parallelism that can be exploited, a focal point of our research that will be elaborated in Section III-B. It is clear that procedures in DL possess abundant parallelism, with larger models exhibiting higher parallelism. Hence, rather than incrementally enhancing accelerator performance, a scale-out approach, by easily expanding computational nodes, can efficiently harness application-level parallelism and rapidly augment accelerator performance for large-scale DL models. In experimental evaluations, our FPGA-based Hydra's 8-card and 64-card prototypes achieve 6.3×-7.5× and 27.7×-55.9× performance improvement in CNNs and LLMs, respectively, compared to a single card. This demonstrates the necessity and effectiveness of exploring application-level parallelism in secure DL reasoning. Specific experimental details are presented in Section V-B.

B. Challenges

1) Architectural Design Weaknesses: Effective utilization of application-level parallelism requires optimized scale-out architecture and communication strategy. However, existing accelerator architectures like FAB [18] struggle with effective scalability and support. FAB connects each FPGA board to a Host CPU, with two FPGAs paired for point-to-point communication via network. Data transfers across multiple boards must pass through the Host, i.e., FPGA to Host (PCIe), Host to Host (LAN), and Host to FPGA (PCIe). In FAB, the diverse communication overhead and reliance on the host for data transfers and synchronization lead to increasing communication costs as the number of cards grows, making it difficult to conceal these overheads. Therefore, a novel and efficient scale-out architecture design is essential for accelerating FHE-based DL inference.

2) Computational Tasks Mapping: In the scale-out architecture, multiple nodes collaborate to perform model inference tasks. This requires task decomposition and mapping, along with data distribution and aggregation. Consequently, the performance of the system is determined by the combined efficiency of computation and communication. Inefficient task mapping will lead to frequent data transfer overheads, severely impacting system performance. In FHE-based DL inference, communication overhead is especially significant due to huge ciphertext size (more than 20MB). Therefore, efficient communication architectures and strategies are needed to reduce overheads. Moreover, DL models, with their diverse network layers and unique computational patterns, require specific task mapping and communication strategies. Additionally, given the significant multiplication depth required for DL inference, bootstrapping operations are essential to refresh the ciphertext from a lower to a higher multiplicative depth, catering to subsequent computational needs. The efficiency of bootstrapping operations affects the performance of the acceleration system. The complexity of bootstrapping makes task decomposition and mapping challenging, necessitating a careful design considering both computational and communication aspects in a scale-out architecture.

To mitigate communication overhead, we establish an independent data transmission unit that can operate concurrently with computation and meticulously devise task decomposition and mapping strategies for all processes within secure DL inference based on the communication paradigm. By optimally distributing the load across multiple nodes and maximizing the overlap between computation and transmission, we can achieve peak system performance within the scale-out architecture. We will elaborate on our design in Section III.

3) Coordination and Synchronization: Synchronization complexities in multi-node collaborative acceleration are multifaceted, encompassing both inter-node computational task synchronization and intra-node communication and computation synchronization. Given the varied tasks across nodes and potential dependencies requiring inter-node communication, the efficiency of the synchronization strategy is crucial, directly affecting system performance. Additionally, intra-node computation and communication also necessitate a synchronization mechanism for effective management due to numerous computation and communication correlations, such as Send After Compute (SAC) and Compute After Receive (CAR). There also exist scenarios where sending is independent of computation and computation is independent of receiving data, thus necessitating the design of appropriate control logic within the node to manage various synchronization situations. Furthermore, these two synchronization mechanisms need to be organically integrated to collaboratively manage the synchronization process of the entire scale-out system. The simplest method is to uniformly manage by the Host CPU, but frequent master-slave communication will inevitably introduce expensive additional overhead, severely impacting performance. Therefore, we need to minimize the involvement of the Host end as much as possible, allowing

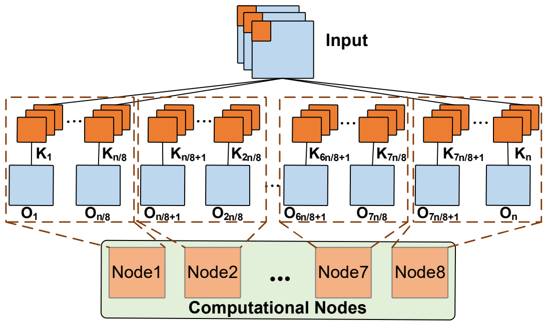


Fig. 1. Computational task mapping of convolution layer.

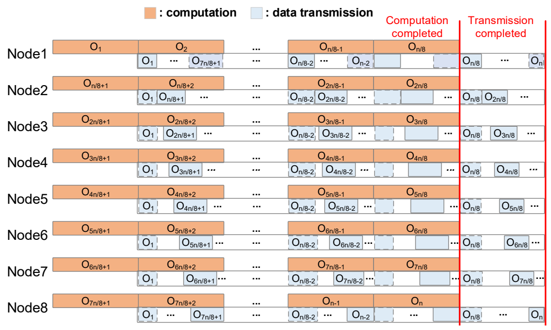


Fig. 2. Inter-node communication of convolution layer.

accelerator nodes to autonomously implement synchronization at the hardware level, efficiently coordinating multiple nodes to complete the overall task. We will elaborate on the design in Section IV.

III. Methodology

A. Key Procedures of CNN and LLM

ConvBN. The BN layer only needs a single HAdd operation post-convolution, not affecting the convolution's computational logic. Thus, ConvBN's task mapping and design are similar to convolution [12]. In FHE-based CNNs, computations across different kernels and input channels are independent, allowing task decomposition at the granularity of computations between distinct kernels on a single ciphertext input. This is shown in Fig. 1 with a single input, n kernels, and 8 nodes. Each node handles $n/8$ kernels, generating $n/8$ outputs, theoretically enhancing computational performance by $8\times$. However, due to the need for comprehensive feature map data in convolutional computation and only partial results available across nodes, data aggregation is needed before moving to the next layer. The hardware architecture separates communication from computation, enabling optimized communication methodologies design. As shown in Fig. 2, upon a subtask completion, the result O_i is sequentially broadcasted to other nodes while each node executes the next subtask. After 8 data transmissions, each node has all other nodes' results. If each computational subtask's delay surpasses that of transmission, the communication delay can be fully overlapped, with only the final subtasks introducing computational overhead. This is easily achieved as each convolution includes several FHE basic operations with a larger computational delay.

FC. The fully connected layer essentially performs the multiplication of the ciphertext vector and the plaintext weight matrix, with frequent rotation operations contributing significantly to the overhead. In the scale-out design architecture, the acceleration of the FC layer hinges on the distribution of rotate operations across multiple nodes and the reduction of communication impact on performance. The vector-matrix multiplication is also a part of the bootstrapping, and we will detail its design in the section III-B.

AvgPool. The average pooling is computed in a similar way to the convolutional layer, where the n-channel input computation also results in n channels, and only the average is computed over a sliding window of size $k \times k$ for each input channel, without the multiple convolution kernels. Therefore, we can regard the averaging operation as a two-dimensional convolution of the input with a convolution kernel with $1/k^2$ values for all elements, and then follow the design of convolution layer, mapping the multiple channels to different nodes and communicating in the same way.

Non-linear. Non-linear functions exist in both CNNs and LLMs, such as ReLU in CNNs, GeLU and Softmax functions in LLMs, and they are usually achieved by evaluating polynomials, which are approximated using the Taylor expansion or the Chebyshev algorithm. Therefore, we can design generic polynomial evaluation algorithms based on multiple accelerated nodes to support both CNNs and LLMs. The algorithm is formalized as Alg. 1, where an instruction like `Card[i].op(NUM=k)` represents placing k tasks of `op` in the task queue of `Card[i]`. The tree-based structure is an optimization method for computing polynomials (shown in Fig. 3(a)). Therefore, the core of acceleration is mapping parts of the linear function that can be parallel computed to multiple nodes and balancing their computational load. First, we determine the depth of the computation tree `tree_depth` based on the current number of cards and the degree of the polynomial. To prevent the computational load on the nodes from being too small and causing the communication overhead to severely impact performance, we do not decompose parts where the degree of the sub-polynomial does not exceed 4, therefore, all nodes will calculate x^2. Next, we assign the tasks of calculating $x^{2^{j+1}}$ to nodes with smaller numbers to balance the computation tree. Subsequently, we distribute the shared computation tasks to each node and assign the communication tasks receiving from the previous step to nodes with larger numbers. Since the computation and communication of each computation node can be parallel, the communication overhead of this process will be masked by the computation. Finally, we aggregate all results to node_0 in a tree-like manner. Based on this strategy, we can determine the optimal computation path for any linear function. Additionally, this type of computation is also included in the bootstrapping process, which will be detailed with examples in Section III-B.

Attention. The attention layer is the core module of LLMs, including PCMM, CCMM and non-linear function evaluation. Non-linear functions can be uniformly implemented using the algorithms described above. In [13], PCMM is implemented by first copying individual input values from a ciphertext matrix

Algorithm 1 Polynomial evaluation of task mapping on multiple acceleration nodes

```
Given: Card 0,1,...$C_{n-1}$ Polynomial with degree of $deg$
  For simplicity, assume n is a power of 2;
  poly_depth := $\lceil \log 2(deg+1) \rceil$;
  card_depth := $\log 2n$;
  tree_depth := min(poly_depth-2,card_depth);
  card_num := $2^{tree\_depth}$;
  for $i = 0$ to $card\_num - 1$ do
      Card[i].multiply(NUM=1);                    ▷ Calculate $x^2$
      for $j = 1$ to $poly\_depth - 2$ do
          if i $< 2^{tree\_depth-j}$ then
              Card[i].multiply_and_send(NUM=1);
          end if                                  ▷ Calculate $x^{2^{j+1}}$
      end for
      k := poly_depth - tree_depth - 2;
      Card[i].add_and_multiply_const(NUM=$2^{k+1}$);
      for $j = 0$ to $k$ do
          if $2^{tree\_depth-j-1} < i < 2^{tree\_depth-j}$ then
              Card[i].receive(NUM=1);
          end if
          Card[i].multiply_and_add(NUM=$2^{k-j}$);
      end for                                     ▷ Common part
  end for
  for $l = 0$ to $tree\_depth - 1$ do
      card_num = card_num / 2;
      for $i = 0$ to $card\_num - 1$ do
          Card[i+card_num].receive(NUM=1);
          Card[i+card_num].multiply_and_send(NUM=1);
          Card[i].receive_and_add(NUM=1);
      end for
  end for                                         ▷ Add up
```

into the entire ciphertext by rotation, then multiplying these ciphertexts by the entire rows of the plaintext matrix, and finally accumulating these intermediate results to complete the process. The implementation of CCMM is much simpler, by packing one of the ciphertext matrices by rows and multiple columns of the other matrix, and then multiplying them once as well as by multiple rotations. Since PCMM and CCMM contain a large number of independent computational tasks as described above, we only need to distribute these tasks evenly across multiple computing nodes, which is a simple and efficient strategy.

FFN. The feed forward network, situated within the LLM network following the attention layer, comprises two fully connected layers and a non-linear function evaluation. These components are calculated in a manner similar to that used in CNNs, thereby enabling their implementation in a similar fashion.

B. Bootstrapping

As shown in Fig. 3 (b), Bootstrapping has three computation procedures, i.e., CoeffToSlot (C2S), Modulus Reduction (MR), and SlotToCoeff (S2C). C2S and S2C perform transition between coefficient and point-value presentation by homomorphically evaluating the DFT on the encrypted data. MR subdivides into EvaExp and Double-Angle Formula (DAF), where EvaExp approximates the cos and sin function, and DAF completes the final modulus reduction based on EvaExp results. While DFT and EvaExp are computationally complex and offer considerable parallel computing opportunities, thus becoming our primary focus.

DFT. As shown in Fig. 3 (c), DFT involves multiple butterfly operations, each being a multiplication between a ciphertext vector and a plaintext sparse matrix. Essentially, DFT is the multiplication of the input ciphertext with several matrices. For a ciphertext of length N, DFT requires log_2N iterations, i.e., log_2N matrix-vector multiplications, each consuming one multiplication depth. This is the basic computational pattern of DFT, i.e., $Radix = 2$, meaning that the input to the basic computational unit is 2. Different $Radix$ correspond to different variants of DFT. As shown in the lower part of Fig. 3 (c), when $Radix = 4$, the computational unit is 4, fusing two iterations of $Radix = 2$ into one, reducing the number of iterations from log_2N to log_4N. Notably, each iteration can have different Radix. In ciphertext-based DFT, a larger Radix implies higher computational overhead (mainly due to more rotate operations) and less consumption of multiplication depth (due to matrix merging and iterations fusion). This is a trade-off at the algorithmic level. For the vector-matrix multiplication (same as the FC), the Baby-Step Gaint-Step (BSGS) algorithm [34] can reduce the computational complexity of Rotate from $O(N)$ to $O(\sqrt{N})$ by reusing some of the rotation results. The naive implementation shown in the upper part of Fig. 3 (d) is to multiply the ciphertext with the diagonal elements of the matrix and then accumulate them. However, because of the requirement for data order in each vector multiplication, we need to rotate the ciphertext in sequence 1, 2, 3 times. BSGS divides the calculation into Giant-Step (gs) and Baby-Step (bs), where bs completes a small amount of ciphertext rotation (rotation 1 in the figure), and gs reuses the initial ciphertext (rotation 0) and the ciphertext rotated 1 step in bs, respectively replacing the ciphertext originally needed to be rotated 2 and 4 steps for calculation. Since the difference between 0 and 2, 1 and 3 is 2, the result of the gs only needs to rotate 2 steps at the end. Compared with the original method, all calculations of gs has only one rotation operation, reducing the total rotation number to $bs+gs$, where $bs \cdot gs = 2Radix$.

There are two points to note about task allocation in DFT on multiple acceleration nodes: (1) DFT includes one bs and multiple gs, and the rotation results in bs are used in all gs. When bs is distributed across multiple compute nodes, all results need to be aggregated to each node before gs is executed, which is inefficient in poor communication environments, and it is better for all nodes to perform the rotation in bs uniformly; (2) simply aggregating data from multiple nodes into a single node will cause significant communication delay when summing the gs results. To fully utilize communication bandwidth, the data aggregation should be in a tree-like pattren. As shown in Fig. 3 (d), taking 4 nodes as an example, the calculation delay of bs is marked in brown, all the nodes

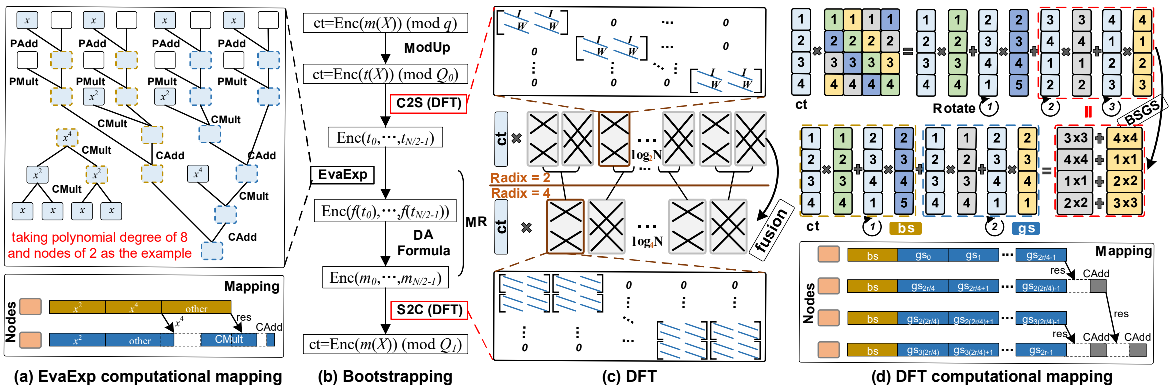


Fig. 3. Bootstrapping process and tasks mapping strategy. DFT and EvaExp are the parts of bootstraping that need to be analysed and designed with focus.

are same. After completing gs marked in blue, the data is aggregated in a tree structure, followed by $log_2 4 = 2$ CAdd completing the DFT.

There are three parameters in DFT: $Radix$, bs and gs. Changing the $Radix$ will affect the overall computational overhead. The performance of bs on multiple nodes is limited and will affect gs that the key to accelerate DFT. Therefore, the optimal algorithmic parameter may not be applicable to the accelerator system. We model the performance of DFT under multiple acceleration nodes, formalized in Eq. 1. gs_s signifies the allocation of gs steps to each computational node following an average manner in a matrix-vector multiplication computation. Here, r is the $Radix$, C_n is the count of accelerated nodes, and b is the rotate operations in the bs step. T represents the time overhead for each operation type. T_{bs} and T_{gs} denote the computation times for the bs and gs steps in each accelerated node, respectively, where T_{gs} encompasses multiply, accumulate and rotate calculation corresponding to PMult, HAdd and Rotation. The process of data accumulation across nodes involves inter-node data transmission, thereby comprises both computation and communication overhead, represented by T_{acc}. Given the use of a cumulative tree structure for optimizing communication efficiency, $log_2 C_n$ rounds of communication and HAdd computation are required. Finally, the total execution time of DFT, denotes T_{DFT}, is the total overhead of $level$ matrix-vector multiplications with $Radix$ is r_i. Based on the model, we can obtain the optimal performance with different parameter settings. We will detail the parameter selection in Section V-G.

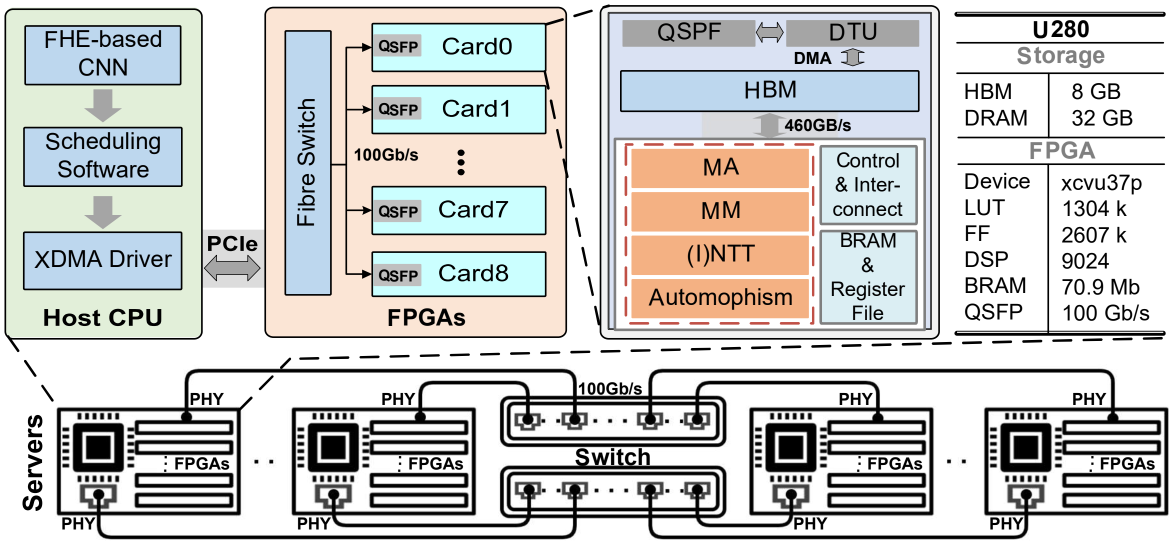
$$\begin{cases} gs_s = 2r/(C_n * b), \\ T_{bs} = b * T_{rot}, \\ T_{gs} = (b * T_{pmult} + (b-1) * T_{hadd} + T_{rot}) * gs_s, \\ T_{acc} = (gs_s - 1) * T_{hadd} + (log_2 C_n + 1) * T_{com}, \\ T_{DFT} = \sum_{i=1}^{level} *(T_{bs} + T_{gs} + T_{acc})_{r_i}. \end{cases} \tag{1}$$

EvaExp. After the C2S operation, the exp function is implemented by evaluating a polynomial with a degree of 59, and is essentially the same as ReLU. For ease of explanation, we illustrate the process with a small polynomial with a degree of 8, as shown in Fig. 3 (a). The computational graph contains three main types of FHE operations: HAdd, PMult, and CMult. CMult has a much higher overhead, hence the key is to balance the number of CMults among the computing nodes. If splitting the tree into four subtrees, two of which contain only one PAdd and one PMult with a small computation but require additional communication of its results. Therefore, we should only split the computation tree into two subtrees (as marked in brown and blue in Fig. 3 (a). At this point, the computational cost of the two parts is unbalanced, requiring 2 and 4 CMult respectively. We thus can assign the calculation of x^4 to the subtree with less CMult, and then send it to the other node, thereby balancing the computational load. Fig. 3 (a) shows the computation mapping strategy of this process. Two nodes are employed, and each node completes 3 CMults, cooperating with two times data transfers to speed up the function. In this manner, we can adeptly maximize the obfuscation of communication's impact on performance.

IV. Hydra Architecture

A. Overall Architecture

The overall architecture of Hydra is shown in Fig. 4. Hydra supports multi-server scaling, with each server containing multiple homogeneous FPGA accelerator cards, interconnected via the high-speed network switch. Each FPGA card primarily includes the computational units (i.e. NTT, MA, MM, and Automorphism), the memory system (high bandwidth memory (HBM) and scratchpad), and the data transfer unit (DTU). All the computational units (CUs) have their own buffer to store the FHE polynomial data of FHE from the off-chip memory HBM. In each cycle, 512 operands are loaded in the CU and written to the output buffer after computation. We use multiple BRAMs and register files as scratchpads to cache the intermediate results and support the parallel read/write of the required data for each CU. The DTU contains the NIC hardcore and control logic, connecting the HBM and the network port transceiver of the FPGA. DTU transmits data independently of the CU via the DMA controller and works with the control and forwarding of the network switch to implement data communication between multiple FPGA accelerator cards. Since all FHE operations can be decomposed into four basic operators corresponding to four CUs, each accelerator card can fully support the FHE acceleration by



U280	
Storage	
HBM	8 GB
DRAM	32 GB
FPGA	
Device	xcvu37p
LUT	1304 k
FF	2607 k
DSP	9024
BRAM	70.9 Mb
QSFP	100 Gb/s

Fig. 4. Hydra overall architecture. Hydra supports multi-server scaling, with servers interconnected by switches and each server containing multiple homogeneous FPGA cards, also connected by switches based on QSFP port. Communication and identification between multiple cards are facilitated through MAC addresses.

combining basic operators. This allows tasks to be decomposed and allocated at any granularity in FHE applications.

B. Computation and Communication Module

Computation. Many works are devoted to the microarchitecture of basic FHE operators, and the key operator, such as NTT, is also maturely designed. To improve the throughput of the CUs with limited FPGA resources, we adopt the Radix-based NTT design with efficient hardware utilization and automorphism, which only requires logical control of data read and write proposed in Poseidon [19]. As the polynomial length used in FHE-based DL is typically 2^{16}, we use Radix-4 instead of the Radix-8 NTT in Poseidon, as it is a better match to the application parameters. For the simpler MM unit, we implement it based on the widely used Barrett algorithm [35]. Moreover, constrained by limited on-chip memory resources and large ciphertexts, Poseidon has no efficient caching strategy, requiring frequent access to HBM, which significantly impacts performance. We adopt the optimization strategy of on-chip memory in MAD [22], using a smaller scratchpad to provide efficient memory access for the CUs.

Communication. The DTU is a critical module that facilitates communication between accelerator cards. It primarily consists of the NIC and control circuitry. The control circuitry interprets data transfer instructions and configures the DMA controller and NIC to conduct the data transfer process. DTU, in conjunction with network switches, supports two inter-card communication modes, point-to-point communication, and broadcast communication, accommodating the demand of FHE-based DL inference under scale-out accelerator system. Point-to-point communication enables an accelerator card to send data to any of the others and vice versa; broadcast communication supports the FPGA card to send data to all of the other cards simultaneously, and after broadcasting, the data is uniformly held by all FPGA cards. Initiating a communication process involves three steps: ❶ DTU parses

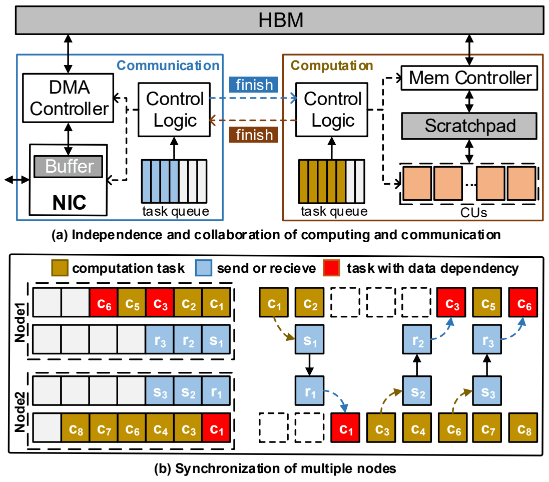


Fig. 5. The synchronization of computing and communication among multiple nodes.

the instructions and configures the DMA controller. ❷ The sender reads data from HBM to NIC via DMA, and sent to the network switch, and forwarded to the destination accelerator card. ❸ The receiver receives the data via NIC and writes to the specified location of HBM through DMA. Moreover, each server runs a runtime program that monitors the operation of the accelerator card and communicates through the switch to synchronize the information.

C. Synchronization Mechanism

The scale-out accelerator system requires inter-server and inter-node synchronization. Server synchronization is rarely required and can be efficiently achieved via the internal network, while card synchronization efficiency has a direct impact on accelerator system performance and must be carefully designed. As shown in Fig. 5 (a), each node incorporates independent communication and computation parts, each with task queue registers and control logic for managing tasks. We

Procedure 1 The synchronization of computation and communication among multiple nodes

```
Communication:
(1) GetTask⟨$t$⟩; fetch a task from the queue, and if the queue is empty, transmission tasks are fully completed
$t \in s$:
(2) Check; check the finish signal from computation unit and ready signal from nodes receiving data
(3) Exe(t); send the data to receiving nodes
(4) Return(1); perform the next communication task
$t \in r$:
(5) Cfg; config the DMA controller ready for receiving data
(6) Signal; send ready signal to the nodes sending data
(7) Wait; wait for the data from computing module
(8) Exe(t); receive the data from sending nodes
(9) Signal; send the finish signal to the computing module
(10) Return(1); perform the next communication task
Computation:
(1) GetTask⟨$c$⟩; fetch a task from the queue, and if the queue is empty, computation tasks are fully completed
$c \in CT_I$:
Exe(c); execute the computation task immediately
(2) Signal; send the finish signal to the communication module
(3) Return(1); perform the next computation task
$c \in CT_d$:
(4) Wait; waiting for the data from communication module
(5) Exe(c); execute the computation task
(6) Signal; send the finish signal to the communication module
(7) Return(1); perform the next computation task
```

categorize computation tasks into data-dependent (CT_d) and data-independent (CT_i). CT_d tasks need to wait for the data reception completion signal from the communication control logic to drive execution. Similarly, communication tasks that depend on computation results need a completion signal to trigger. This achieves intra-node synchronization, and based on this design, inter-node synchronization can be simplified to communication synchronization. Since receive and send tasks always appear in pairs and their order is predetermined, we can adopt a simple handshake mechanism to implement the synchronization function. The specific process is that when a receive task is taken from the queue, the DMA controller and NIC are configured without waiting, and a ready signal is sent to the data sender after configuration. Upon receiving the ready signal, the data sender records it for a handshake check before executing the send task. As communication tasks in different nodes strictly correspond and only the next task is loaded after the current one is completed, this ensures correct communication order. Details of the synchronization of computation and communication between multiple nodes are shown in Procedure 1. Task completion is indicated when both the communication and computation task queues are empty, and then the accelerator node signals the Host CPU, which

Procedure 2 Scheduling strategies between multiple cards in multiple servers

```
Given: Task 1,2,...$T_n$; Server 1,2,...$S_n$; Card 1,2,...$C_n$;
  while $i < T_n$ do
    for $j < S_n$ do                                    ▷ fully parallel
      Server[j].get(Task[i].subtask[j]);
      for $k < C_n$ do                                  ▷ fully parallel
        Card[k].exe(Task[i].subtask[j][k]);
        Card[k] finished;
      end for
      Check Card[1-$C_n$] finished;
      Server[j] finished;          ▷ all the Cards finished indicates the server finished
    end for
    Check Server[1-$S_n$] finished;
    Task[i] finished;              ▷ all the Servers finished indicates this task finished
  end while
```

runs a monitoring program.

Fig. 5 (b) illustrates an example of two-node synchronization. For simplicity, we assume that the delays of computation tasks and communication tasks (marked in brown and blue respectively) are the same. Communication tasks are divided into sending (s) and receiving (r), and tasks that depend on the other nodes' results (CT_i) are marked in red. As shown in the figure, tasks $c1$ and $c2$ in the task queue of node1 have no data dependencies and can be executed directly, while the first task in node2 depends on task $c1$ in node1, so it needs to wait. When node1 completes $c1$, it triggers task $s1$ in the communication queue to send the result of $c1$ to node2. Task $r1$ in the task queue of node2 is responsible for receiving the data sent by node1 and triggers the task in the computation task queue after the reception is completed. At this point, the computation tasks in node2 have no data dependencies and can be executed continuously. The third and fifth tasks in node1 depend on tasks $c3$ and $c6$ in node2, respectively, so before the execution they need to wait for node1 to complete the corresponding tasks and drive $s2$, $r2$, and $s3$, $r3$. Finally, when the task and communication queues are both cleared, the node will signal the host CPU, representing the completion of tasks.

D. Scheduling Strategy and Dependence Management

Hydra supports multi-server expansion, with each server housing multiple FPGA accelerators. We necessitate the collaboration of all computational nodes to accomplish the computational task. Procedure 2 illustrates the scheduling strategy under multiple servers, with a total of T_n computational tasks and S_n servers, each server containing C_n FPGA accelerators. The data dependencies for specific model are pre-determined based on the methods in Section III and managed by host-side scheduling software, which organize tasks at the CNN/LLM step level (e.g., Conv, Boot, Attention), and each server is responsible for subtasks within it. Before accelerator startup, the scheduling software running in each server preloads data and task instructions onto each FPGA, and data parallelism and dependencies within the task are embedded in various data transfer and computation task instructions (shown in Figure 5). With Hydra's task synchronization mechanism, computational and communication dependencies across multiple cards are automatically managed on the FPGA side. The execution is parallel between the servers as well as the respective FPGAs. Once an FPGA completes its own allocated subtask, it sends a completion signal to the host's runtime monitoring program. When all FPGAs on a server finish, the subtask is complete. Once all servers finish, the overall task is complete and servers signal the start of the next task. Note that synchronization between servers only requires sending a completion signal, with negligible overhead. Additionally, tasks are managed as instructions, allowing multiple tasks to be loaded into each FPGA's task queue at once. Subsequent tasks can be added before the FPGA completes the current one, preventing idle time from host task allocation.

V. Evaluation

A. Experimental Setup

Platform. Hydra is a practical scale-out FHE accelerator, so we build a real-world experimental environment based on the x86 CPU system. The Hydra accelerator is implemented in a high-performance computing server, equipping eight Xilinx Alveo U280 FPGAs [36] and multi-card networking through high-speed switches. We employ our customized FHE library tailored to the accelerator hardware to organize and control the execution of applications on Hydra.

Baseline. In our experiments, we compare Hydra with the state-of-the-art FPGA solution Poseidon [19] and FAB (FAB-S: single card, FAB-M: 8 cards) [18], and 4 FHE accelerator ASICs, e.g., Craterlake [33], ARK [15], BTS [16], and SHARP [21]. To evaluate the performance of the scale-out architecture, we establish the three prototypes of Hydra, namely Hydra-S (1 server with 1 FPGA card), Hydra-M (1 server with 8 FPGA cards), and Hydra-L (8 servers with 64 FPGA cards). The hardware architecture of Hydra-S is identical to the compute node of Hydra-M and Hydra-L but without the DTU. We also evaluate the performance of FAB's proposed multi-card architecture with 8 cards (FAB-M) and 64 cards (FAB-L) for a scalability comparison with Hydra. In FAB, we assume a PCIe bandwidth of 16GB/s (supported by the Xilinx Alveo U280 FPGA) and a 10Gb/s high-speed LAN connection.

Benchmark. *(1) ResNet-18.* ResNet-18 trained with the ImageNet dataset (224×224) [37], it has 18 convolution layers and one fully-connected layer. The activation functions are uniformly ReLU. We implement it based on the method proposed by [12], and use it to evaluate the performance of Hydra. Following the implementation, encrypting the input data requires 2 ciphertexts.

(2) ResNet-50. ResNet-50 has larger convolution parameters as well as deeper model depth compared to ResNet-18. Its implementation approach is identical to [12], [30], and also uses the ImageNet dataset [37]. We chose it as a benchmark to evaluate the performance of Hydra with a larger-scale CNN model.

(3) BERT-base. BERT-base is a classical large language model of enhancing the efficacy of natural language processing tasks. Its FHE-based implementation [13] is specifically designed to optimize for the PCMM, CCMM, Nonlinear functions and bootstrap insertion. The size of the input sequence is 128×768. We selected it as the benchmark to evaluate Hydra's performance on LLM.

TABLE II
Full-system performance comparison with SOTA accelerator prototypes. We use the actual benchmark execution time in ``s'' as the metric.

	ResNet-18	ResNet-50	BERT-base	OPT-6.7B
		ASIC		
CraterLake [33]	5.51	89.76	76.34	2615.11
BTS [16]	32.81	534.06	454.23	15560.30
ARK [15]	2.15	34.95	29.73	1018.34
SHARP [21]	1.70	27.68	23.54	806.53
		FPGA		
FAB-S [18]	131.94	2255.46	1302.68	51813.24
Poseidon [19]	55.05	915.51	616.59	24006.44
FAB-M [18]	18.89	287.27	208.54	6841.11
Hydra-S	**41.29**	**686.63**	**462.44**	**18004.83**
Hydra-M	**5.60**	**86.79**	**72.31**	**2382.18**
Hydra-L	**1.49**	**12.94**	**13.81**	**321.58**

(4) OPT-6.7B. There is no existing implementation of OPT-6.7B that has a similar model structure to BERT-base and a larger network size compared to it, i.e., both the number of layers and attention heads is 32, where the embedding size is 4096. The size of the input sequence is 200×4096. We implement the model based on the methodology of [13] to evaluate the performance of Hydra on larger LLMs.

In this section, we evaluate the performance of the key procedure of DL and full-system in Fig. 6 and Table II. Note that Hydra uses parameters log(PQ)=1692 and logQ=1260 (same as SHARP [21]), while other accelerators use their own respective parameters (e.g., FAB: log(PQ)=1728, logQ=1242). These variations affect the internal computations of FHE operations but not their functionality. The model's computational process follows the optimized implementation in [12], as shown in Table I.

B. Performance

1) Full-system performance: The experiment compares the full-system performance of Hydra with SOTA accelerator prototypes including FPGA and ASIC. The data is sourced from precise simulations based on the specific architectures of these accelerators. As shown in Table II, compared to ASIC accelerators, Hydra-M's performance significantly surpasses BTS, and it achieves performance close to that of CraterLake. As for Hydra-L, the multi-server expanded prototype, its performance exceeds all existing SOTA ASIC accelerators (1.14× to 2.5×) on both CNNs and LLMs. Compared with the SOTA single FPGA solution Poseidon and FAB-S, Hydra-S exhibits superior performance across four DL models. Compared to FAB-S, Hydra-S has a greater performance improvement (2.8× to 3.1×), and there are about 1.3× improvements over Poseidon. This is due to the adoption of cache-friendly data flow planning methods in MAD. FAB-M is an 8-card architecture proposed by [18], and since FAB-M does not have a mapping strategy for designing tasks for DL inference and does not support multi-card acceleration of bootstrapping, we apply Hydra's task mapping strategy to FAB-M to make

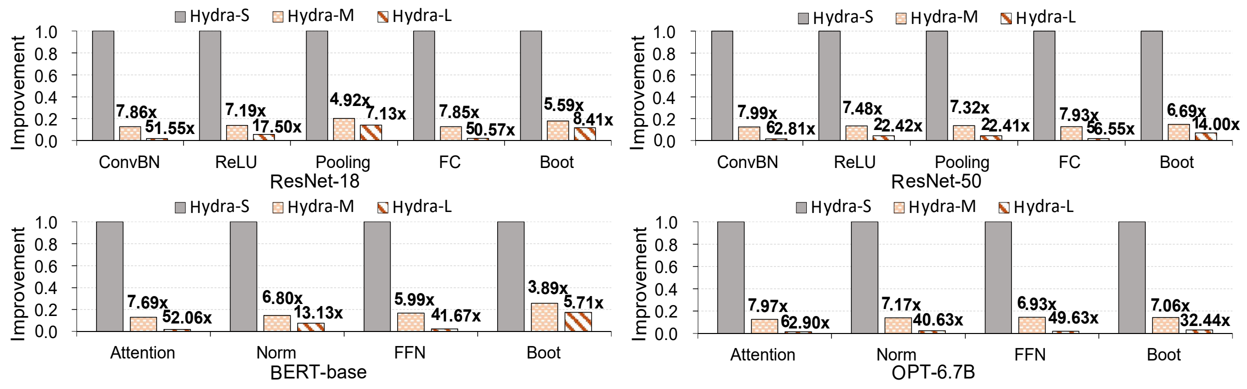


Fig. 6. The performance improvement of key procedures of four DL benchmarks on Hydra-S, Hydra-M and Hydra-L, normalized to Hydra-S.

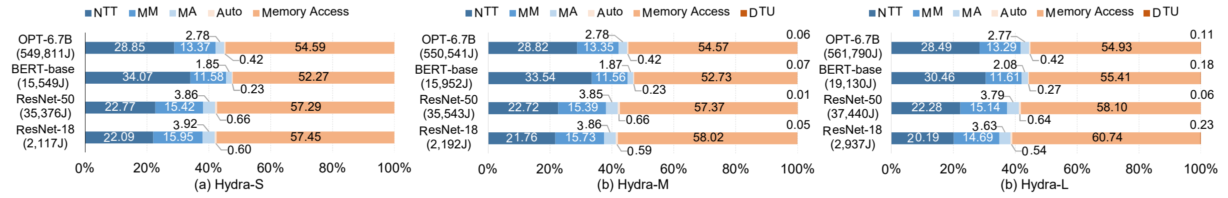


Fig. 7. Full-system energy consumption and breakdown of selected DL models on three prototypes of Hydra.

a fair comparison of the architectures. As shown in table II, Hydra-M improves performance by 2.8× to 3.3× compared to FAB-M. This is mainly due to the inefficiency of FAB-M's architecture in inter-card communication. In FAB-M, each of the eight FPGA boards is connected to a host CPU, and the boards are connected in pairs, this setup only supports efficient point-to-point communication between fixed pairs of FPGAs. Communication between multiple FPGAs requires data transfer through the host CPUs, which introduces significant communication overhead (PCIe and network data transfers). In contrast, Hydra's multi-card interconnect structure and synchronization mechanism efficiently reduce the data transfer overhead, resulting in better performance than FAB-M. Furthermore, compared to Hydra-S, both Hydra-M and Hydra-L achieved significant performance improvements, with Hydra-M showing an increase of 6.3× to 7.5× and Hydra-L demonstrating an enhancement of 27.7× to 55.9×. This underscores Hydra's excellent scalability.

2) Key Procedure Performance: We also observe the performance improvement of different key procedures of DL inference on three prototypes of Hydra. As shown in Fig. 6, for CNN models, Hydra-M demonstrates substantial speedup across all key procedures, achieving over 7× speedup in ConvBN, ReLU, and FC, and surpassing 5× in Pooling and Boot. This significant performance gain is attributed to the high, yet underutilized, parallelism in ResNet-18 and ResNet-50 with 8 cards. Conversely, Hydra-L exhibits exceptionally high speedup in ConvBN and FC (over 50×), though the acceleration in ReLU, Pooling, and Boot is more modest. This suggests that the parallelism in these latter operations is relatively limited. For LLMs, the attention and FFN, analogous to ConvBN in CNNs, exhibit high performance improvements under both Hydra-M and Hydra-L, with further node expansion to yield even greater performance enhancements. However, the performance of Norm and Boot in BERT-base is constrained by the smaller scale of BERT-base, which limits the extent of parallelizable computations. In contrast, the slightly larger OPT-6.7B showcases impressive performance. Although the parameter size of 6.7B is still relatively small for LLMs, it is able to effectively harness the architectural benefits of Hydra, with both Hydra-M and Hydra-L achieving up to 7× and 40× speedup in Norm and Boot, respectively, compared to Hydra-S, demonstrating the scalability and efficiency of the Hydra architecture.

TABLE III
EFFICIENCY ANALYSIS. WE USE ENERGY DELAY AREA PRODUCT (EDAP) AS THE METRIC. LOWER IS BETTER.

	ResNet-18	ResNet-50	BERT-base	OPT-6.7B
CraterLake [33]	1.40	371.4	268.7	315,260
BTS [16]	53.81	14,257.4	10,313.9	12,103,166
ARK [15]	0.54	143.7	104.0	122,024
SHARP [21]	0.09	22.8	16.5	19,330
Hydra-S	**0.12**	**32.8**	**8.8**	**12,703**
Hydra-M	**0.15**	**33.8**	**12.5**	**13,541**
Hydra-L	**0.59**	**48.1**	**38.1**	**16,208**

C. Energy

Energy Consumption and Breakdown. In addition to performance, we also evaluate the full-system energy consumption of three Hydra prototypes when running the selected benchmarks. As shown in Fig. 7, memory access accounts for the largest share of the energy consumption for all benchmarks under Hydra-S, Hydra-M, and Hydra-L, which reflects the access-intensive characteristic of FHE applications. Among the CUs, NTT and MM are most frequently utilized, thereby taking a large energy consumption, while the energy consumption of MA is minimal due to its simple computing logic. Hydra-M and Hydra-L require communication between accelerator

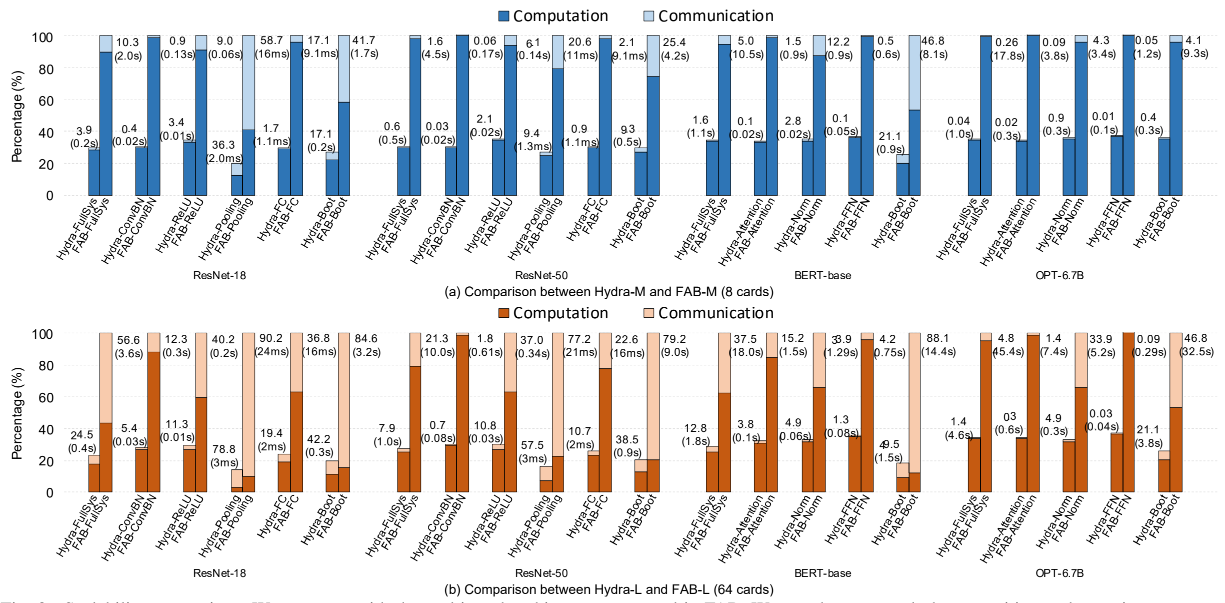


Fig. 8. Scalability comparison. We compare with the multi-card architecture proposed in FAB. We use the same task decomposition and mapping strategy as Hydra for performance evaluation. The upper row denotes the communication and computation overhead of four benchmarks on Hydra-M and FAB-M, and the bottom row denotes that on Hydra-L and FAB-L. The performance is normalized to FAB.

cards, and frequent data distribution and aggregation by DTU introduces additional energy consumption, but only accounting for less than 1% due to the NIC hardcore with low power consumption.

Energy Efficiency. Existing ASIC accelerators do not support multi-card expansion, so to demonstrate the efficiency comparison of the scale-out Hydra with these accelerators, we use EDAP (Energy-Delay-Area Product) as the evaluation metric. We evaluate Hydra's ASIC power consumption and area based on RTL implementation, standardized using 7nm technology. As shown in Table III, Hydra-S exhibits the highest energy efficiency, while Hydra-M and Hydra-L show progressively lower efficiency due to the communication overhead introduced by multi-card setups. Hydra-M's energy efficiency is comparable to that of SHARP and surpasses CraterLake, BTS, and ARK. Similarly, Hydra-L's energy efficiency outperforms all other ASIC accelerators except SHARP, particularly in the LLM OPT-6.7B, where it exceeds all ASIC solutions, i.e., CraterLake (by 19.4×), ARK (by 75×), and SHARP (by 12.2×). This indicates that even though Hydra improves performance through multi-node expansion, it still achieves higher energy efficiency compared to existing ASIC accelerators under the same chip technology.

D. Scalability Comparison

Scalability is at the core of Hydra. As computing nodes are scaled up, optimizing communication overhead becomes crucial for system efficiency. We evaluated Hydra's communication and computation overhead during node expansion with two prototypes, Hydra-M and Hydra-L, and compared them with the multi-card architecture proposed in FAB. We used the same task decomposition and mapping strategy as Hydra for performance evaluation. Unlike Hydra, FAB lacks hardware-based inter-card communication and synchronization, necessitating full utilization of the host CPU for management. As depicted in Fig. 8, Hydra-L demonstrates a lower communication overhead percentage across four benchmarks and key procedures compared to Hydra-M. The significant increase in Hydra-L's communication overhead percentage is primarily due to the substantial reduction in computation overhead as more nodes are added, which lowers the overall overhead. Consequently, even a small increase in communication overhead appears significant in percentage terms. In reality, the substantial reduction in computation overhead with minimal growth in communication overhead during node expansion signifies good scalability. In contrast, FAB's inefficient communication and synchronization result in much higher communication overhead than Hydra. For FAB-L, the communication overhead and its percentage increase dramatically compared to FAB-M, with operations like Boot and Pooling reaching as high as 90%. Additionally, FAB's computational performance is weaker than Hydra's, which means the communication overhead in FAB's multi-card architecture has a more detrimental impact on system performance.

E. Scalability Analysis

As shown in Fig. 9 (a) and Fig. 9 (b), the efficiency of both CNN and LLM models improves with an increase in the number of cards. For ResNet-50, the performance of ConvBN scales faster compared to Boot, indicating that Boot has relatively lower parallelism compared to other procedures. As the number of cards increases, Boot will reach its performance bottleneck sooner. For OPT-6.7B, the performance growth curves of its key procedures are closely matched and maintain a high rate of increase. This is due to the rich parallelism in these procedures, which remains underutilized even with

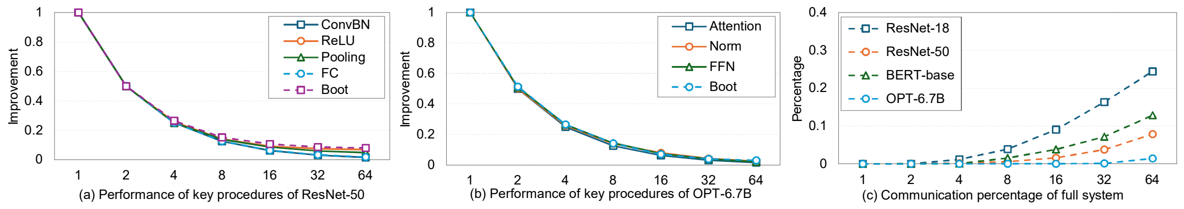


Fig. 9. Scalability analysis of the computing nodes.

TABLE IV
FPGA RESOURCE UTILIZATION OF HYDRA WITH A SINGLE CARD.

Resource	Utilized	Available	Utilization (%)
LUTs (k)	997	1,304	76.5
FFs (k)	1,375	2,607	52.7
DSP	8,704	9,024	96.5
BRAM	3,072	4,032	76.2
URAMs	768	962	79.8

64 computing nodes. As the number of nodes continues to expand, the performance will be further improved significantly.

Fig. 9 (c) illustrates the change in the proportion of communication overhead to total overhead. The communication overhead of ResNet-18 grows most rapidly, while that of OPT-6.7B increases more slowly. As the number of nodes increases, ResNet-18's performance growth will slow and reach its bottleneck sooner. In contrast, OPT-6.7B still has substantial room for performance improvement. This also demonstrates that Hydra's scalability can handle increasingly larger model sizes while maintaining excellent performance.

F. FPGA Resource Utilization

Table IV shows the FPGA resource utilization of Hydra's single card prototype. The DSP has the highest utilization of 96.5%, which is mainly used for NTT, a large number of multiplication computations in the MM cell. the LUT and FF utilization is slightly smaller compared to the DSP, 76.5% and 52.7%, respectively. hydra uses both BRAMs and URAMs. dual-port BRAM is used as a data cache for the CUs to provide read/write usage of 3072 blocks accounting for 76.2% of the total available resources, while only a single-port URAM is used to cache the secret key required for computation at a utilization rate of 79.8%.

G. Parameters Selection

Table V shows the selection of key parameters, i.e., $Radix$ and bs, in the DFT during bootstrapping when the multiplication depth consumption is 3 (as set in [12], [30]). Due to the different usage of ciphertext slots in homomorphic DL inference, we list the parameter settings under different slot numbers. Hydra-S only contains a single accelerator card, and the optimal parameters in terms of the algorithm are also optimal for it. Hydra-M and Hydra-L, on the other hand, have 8 and 64 FPGA cards respectively, so their choice of bs differs from Hydra-S and is chosen to minimize the $bs + \frac{gs}{Card\_num}$. Furthermore, it can be seen that the bs of Hydra-L is smaller than that of Hydra-M. This is because under more computing nodes, a larger gs can exert its parallel computing capability.

TABLE V
THE OPTIMAL CHOICE OF THE PARAMETERS RADIX AND BS UNDER THREE PROTOTYPES OF HYDRA.

logSlots	Hydra-S		Hydra-M		Hydra-L	
	Radix	bs	Radix	bs	Radix	bs
12	(16,16,16)	(4,4,4)	(16,16,16)	(1,2,2)	(8,4,128)	(1,1,2)
13	(16,16,32)	(4,4,8)	(32,16,16)	(2,2,2)	(8,8,128)	(1,1,2)
14	(32,16,32)	(4,4,8)	(32,16,32)	(2,2,2)	(8,16,128)	(1,1,2)
15	(32,32,32)	(4,8,8)	(32,16,64)	(2,2,4)	(8,32,128)	(1,1,2)

H. Discussion

Hydra is a scale-out acceleration architecture designed based on the rich application-level parallelism in secure DL (including CNN and LLM) inference. In addition to the optimized allocation strategy and hardware architecture design, the characteristics of the specific application itself also affect the performance potential of Hydra. For CNNs such as ResNet-18 and ResNet-50, although they have a deeper network layer depth, there are data dependencies between different residual blocks. Hydra can only fully parallelize and accelerate the independent residual network layer without data dependencies, and its performance mainly depends on the parallelism of the layer. For LLMs, the main component is the Transformer module, whose main computational bottleneck is large-scale dense matrix multiplication, rather than cascading a number of small-scale data-dependent layers that do not require more frequent data distribution and aggregation. Therefore, it is easier to exploit Hydra's strengths compared to the ResNet family, which is why Hydra performs better on LLMs compared to CNNs for the EDAP shown in Table III. This is not to say that Hydra is not suitable for CNNs. In fact, the parallelism on independent network layers is also abundant in CNNs, as Table I proves. Thus, Hydra is suitable for a wide range of DL models, particularly those with significant computation and parallelism in data-independent network layers.

VI. CONCLUSION

In this paper, we propose Hydra, the high-performance scale-out accelerator architecture for FHE-based DL inference. We design task decomposition and mapping methods for all major steps (including bootstrapping). By designing independent computation and communication modules and their corresponding synchronization mechanisms, Hydra maximizes the overlap of communication latency and computation latency among computing nodes, thereby achieving excellent system performance. Hydra can support arbitrary node expansion to cope with the computational expansion of large-scale and even ultra-large-scale DL models, proposing a new paradigm for the design of secure DL inference accelerator architectures.

References

[1] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 9466–9471.

[2] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "Pir-psi: scaling private contact discovery," *Cryptology ePrint Archive*, 2018.

[3] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1243–1255.

[4] L. Shen, X. Chen, D. Wang, B. Fang, and Y. Dong, "Efficient and private set intersection of human genomes," in *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2018, pp. 761–764.

[5] A. Al Badawi, C. Jin, J. Lin, C. F. Mun, S. J. Jie, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1330–1343, 2020.

[6] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[7] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[8] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *arXiv preprint arXiv:1811.09953*, 2018.

[9] H. Chabanne, R. Lescuyer, J. Milgram, C. Morel, and E. Prouff, "Recognition over encrypted faces," in *Mobile, Secure, and Programmable Networking: 4th International Conference, MSPN 2018, Paris, France, June 18-20, 2018, Revised Selected Papers 4*. Springer, 2019, pp. 174–191.

[10] T. Ishiyama, T. Suzuki, and H. Yamana, "Highly accurate cnn inference using approximate activation functions over homomorphic encryption," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 3989–3995.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[12] 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," in *International Conference on Machine Learning*. PMLR, 2022, pp. 12 403–12 422.

[13] J. Zhang, J. Liu, X. Yang, Y. Wang, K. Chen, X. Hou, K. Ren, and X. Yang, "Secure transformer inference made non-interactive," *Cryptology ePrint Archive*, 2024.

[14] E. Liberty, Z. Karnin, B. Xiang, L. Rouesnel, B. Coskun, R. Nallapati, J. Delgado, A. Sadoughi, Y. Astashonok, P. Das *et al.*, "Elastic machine learning algorithms in amazon sagemaker," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 731–737.

[15] 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," *arXiv preprint arXiv:2205.00922*, 2022.

[16] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 711–725.

[17] 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*, 2021, pp. 238–252.

[18] 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," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[19] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 870–881.

[20] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[21] 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*, 2023, pp. 1–15.

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

[23] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, 2014.

[24] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 56–64.

[25] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 3–16, 2016.

[26] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-lwe and she cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 1, pp. 157–166, 2014.

[27] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.

[28] A. C. Mert, S. Kwon, Y. Shin, D. Yoo, Y. Lee, S. S. Roy *et al.*, "Medha: Microcoded hardware accelerator for computing on encrypted data," *arXiv preprint arXiv:2210.05476*, 2022.

[29] 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)*. IEEE, 2021, pp. 26–39.

[30] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "Dacapo: Automatic bootstrapping management for efficient fully homomorphic encryption."

[31] DESILO, "Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy," 2023, https://github.com/Desilo/liberate-fhe.

[32] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. K. M. Mi, "Multi-gpu design and performance evaluation of homomorphic encryption on gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 379–391, 2020.

[33] 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 *ISCA*, 2022, pp. 173–187.

[34] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*. Springer, 2018, pp. 360–384.

[35] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[36] Xilinx, "Product brief of smartssd," https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.