2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00107

Hyena: Balancing Packing, Reuse, and Rotations for Encrypted Inference

Sarabjeet Singh
Kahlert School of Computing
University of Utah
Salt Lake City, Utah
sarab@cs.utah.edu

Shreyas Singh
Kahlert School of Computing
University of Utah
Salt Lake City, Utah
shreyas.singh@utah.edu

Sumanth Gudaparthi
Kahlert School of Computing
University of Utah
Salt Lake City, Utah
sumanth.gudaparthi@gmail.com

Xiong Fan
Department of Computer Science
Rutgers University
New Brunswick, New Jersey
leofanxiong@gmail.com

Rajeev Balasubramonian
Kahlert School of Computing
University of Utah
Salt Lake City, Utah
rajeev@cs.utah.edu

***Abstract*—Deep neural networks are widely used in a range of commercial services. Many of these services are hosted on the cloud, requiring users to send their personal data to the cloud. This, in turn, exposes the user's private and sensitive data to several third parties. To address this problem, Homomorphic Encryption (HE) has been introduced, where the user encrypts their data before sending it to the cloud; the cloud performs operations on encrypted data and returns a ciphertext that the user must then decrypt. While this approach keeps user data private, it demands orders of magnitude more computation and data movement. It is, therefore, imperative to design hardware/software techniques to lower the overheads when executing AI services under Homomorphic Encryption schemes.**

In this paper, we consider a range of HE implementations for AI inference and address the key bottlenecks in state-of-the-art frameworks. We start by making the case for a hybrid HE and Multi-Party Computation (MPC) scheme that is more practical than pure Fully HE. This paper introduces new techniques at various levels: (i) we introduce new data packing techniques that result in lower data movement, (ii) we introduce new dataflows that increase reuse and reduce other costly HE operations (rotations, key switching, NTT conversion), (iii) we evaluate Hyena on a balanced pipelined architecture that efficiently handles the above primitives. The resulting framework, Hyena (new packing + dataflow), achieves better performance and energy than several packing baselines. Compared to the widely used Channel-packing, Hyena is $38\times$ faster and achieves $162\times$ lower energy consumption, with an overall ResNet20 inference end-to-end latency of 11.4 *ms*, using a 163 mm^2 accelerator dissipating 16.75 W.

1. Introduction

Modern-day services, ranging from medical diagnosis to facial recognition to assistive services, have become heavily dependent on deep learning. When deployed at scale, these services are typically hosted in large data centers. Users send their data (often private/sensitive) to these remote servers to execute deep neural network (DNN) models. The exposure of private user data to the service provider can lead to widespread data misuse by the provider, rogue employees, or malicious agents that breach the data center infrastructure. This has fueled interest in privacy-preserving deep learning that allows users to leverage the benefits of AI models and cloud platforms without exposing raw data.

Homomorphic Encryption (HE) has emerged recently, leading the charge for providing privacy-preserving deep learning with varied implementations [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. HE schemes allow cloud operators to perform complex calculations over encrypted user data and produce encrypted results, without having any knowledge of the underlying data. The many simple, error-tolerant, and parallel operations in DNNs make them a perfect fit for HE. However, HE imposes a significant burden on compute operations, data movement, complexity, and memory capacity. State-of-the-art implementations of HE for deep learning remain orders of magnitude slower than their plaintext counterparts [1], [3], [4], [12], [13].

Many efforts have been made to execute HE-based computations efficiently. BTS [12], CraterLake [13], and ARK [14] employ Fully Homomorphic Encryption (FHE), i.e., they perform all computations on the server and invoke Bootstrapping when the HE noise exceeds a threshold. On the other hand, Cheetah [1], Cheetah-2PC [6], and Gazelle [2] use frameworks where the client and server exchange data after every DNN layer via a Secure Multi-Party Computation (MPC) protocol. While this entails high communication costs, it avoids Bootstrapping and dramatically reduces the overall compute burden on the server. Our analysis shows that Cheetah-2PC is currently the best overall design point. We also observe that the data encoding scheme in these works leads to several inefficiencies: large data movement on the

DOI 10.1109/SP54263.2024.00107

server, compute under-utilization, complex permutations, and frequent data encoding conversions.

We design a new HE framework called *Hyena* that addresses all of the above problems. ➀ The foundational modification is a more compact data encoding scheme, aka *packing* – the framework is therefore named after a pack animal. The new data packing has several cascading effects. It reduces data movement from the server's external memory. It also leads to better utilization of compute units. ➁ We then combine the packing with a dataflow (ordering of computations) that increases data reuse and decreases the required computations. While DNN dataflows are well studied, HE algorithms introduce new artifacts that must be considered for each packing and dataflow. To aggregate partial sums, each partial sum has to be rotated using format conversion, shift/permute, and key switching. The Hyena packing and dataflow improve upon previously studied packings on all these metrics. ➂ Finally, we evaluate our proposal on a hardware accelerator with a custom pipeline that is tailored to execute all of the above operations efficiently. Note that while we advocate for a Hybrid HE-MPC setup, Hyena's techniques (packing, dataflow) are server-side optimizations and are applicable to Fully HE works as well.

In brief, the paper makes the following contributions:

- The paper studies the trade-offs between FHE and hybrid HE+MPC approaches and makes a case for the latter for practical privacy-preserving inference.
- This work compares several prior HE-packing schemes - LoLa [15], Channel [2], Cheetah-2PC [6], CryptoNets [3], and many more. We observe that high packing density can improve several system metrics, but it requires a dataflow that keeps rotation overheads under control.
- We co-design a new data packing scheme and dataflow. This has several effects: it impacts reuse and data movement from memory, it improves compute utilization, and it reduces invocations of expensive HE primitives (format conversion, shifts, permutations, key switches).
- We evaluate over a balanced pipelined accelerator that has tailored resources for each dataflow. We observe that the combination of packing/dataflow techniques yields a 38× speedup and 162× lower energy relative to baseline Channel-packing, which is adopted by state-of-the-art implementations [12], [13], [14]. As a result, we quantify ResNet20 private inference with an end-to-end latency of 11.4 *ms* on a 163 mm^2 chip.

The paper is organized as follows. Section 2 presents a brief background about Homomorphic Encryption and its operations. Section 3 sets up the framework for encrypted inference using Homomorphic Encryption. Section 4 analyses the state-of-the-art HE-based CNN work and identifies its bottlenecks. Section 5 presents our novel Hyena packing and dataflow that addresses these issues. Section 6 details the implementation of our architecture. Finally, we evaluate our contributions in Section 7, compared against several baselines.

2. Background: HE Preliminaries

Homomorphic encryption enables computation over encrypted data, first shown possible by Gentry [16]. Since then, there have been many efforts to provide additional functionalities and improve performance [17], [18], [19], [20], [21], [22], [23]. As a result, popular schemes – like BGV [21], B/FV [18], [20], GSW [19], FHEW [24], TFHE [25], and CKKS [22] – have emerged and found adoption in startups [26] and established companies like CryptoExperts [27] and the Microsoft SEAL [28] library. While the schemes differ in how they encrypt the data, they all derive their security from the hardness of the Ring-LWE problem [29] and provide the same set of operations over ciphertexts, which are represented as *polynomials*. We use CKKS as it allows computations over floats. We now review some preliminaries for FHE.

2.1. Homomorphic Operations' Complexity

HE allows addition and multiplication over ciphertexts. To provide such functionality, HE schemes encrypt plaintext using lattices, most commonly relying on the Ring-LWE assumption. The plaintext message, *a single* integer (m) modulo q, is first encoded as a polynomial pt (specific encoding strategies described later). The coefficients of the polynomial form a vector and all operations are performed over these vectors. The ciphertext (ct) is a *pair* of polynomials of degree $N-1$ with coefficients modulo Q. To form the ciphertext, a uniformly random polynomial $a \in \mathbb{R}_Q$ is sampled and multiplied with the secret key $s \in \mathbb{R}_Q$. Plaintext is then hidden by adding to the multiplicative result and obscuring with some error e. The ciphertext is computed as:

$$\mathbf{ct} = (a, b = a \cdot s + pt + e)$$

All ciphertext computations in HE are performed modulo Q, and polynomial multiplication is modular multiplication, which keeps the output degree and the coefficient bit-width constant.

The decryption calculates $(b - a \cdot s) \mod Q$. For correct decryption, the error e must be small. During multiplication operations on ciphertext, noise increases *multiplicatively*, while addition increases the noise additively. To ensure correct decryption, the ciphertext has a certain noise budget, typically a depth factor that bounds the number of homomorphic operations, especially multiplications, that can be performed over the ciphertext. To maintain a reasonable depth and guarantee security, large HE parameters (N, Q, etc.) are required, which in turn causes orders of magnitude slowdown compared with unencrypted data computation – *HE based DNN inference is 2-5 orders of magnitude slower than unencrypted inference accelerators* [1], [3], [30], [31], [32]. The Residue Number System (RNS) representation [33] is used to reduce the overheads of wide arithmetic and is assumed throughout this paper.

2.2. Efficient PolyMult with NTT

While polynomial addition (PolyAdd) simply requires coefficient to coefficient addition, multiplying two polynomials (PolyMult) requires convolving their coefficients, an expensive $O(N^2)$ operation. Similar to the Fast Fourier Transform for convolution, Number Theoretic Transform (NTT) [34] is used to perform polynomial multiplication faster. Degree-$(N-1)$ polynomials are first transformed from their original *coefficient domain* to an *evaluation domain* using the $O(N \log N)$ NTT algorithm, followed by an $O(N)$ coefficient-wise multiplication. Then, the result is reverted to the coefficient domain using the Inverse-NTT ($O(N \log N)$). The evaluation domain is particularly convenient as it converts $O(N^2)$ convolution in PolyMult to simple element-wise $O(N)$ multiplication. Moreover, element-wise addition in evaluation domain also results in element-wise addition in coefficient domain, summarized as:

$$\mathsf{NTT}(ab) = \mathsf{NTT}(a) \cdot \mathsf{NTT}(b)$$

$$\mathsf{NTT}(a+b) = \mathsf{NTT}(a) + \mathsf{NTT}(b).$$

Polynomials are kept in evaluation domain for as long as possible to avoid the $O(N \log N)$ cost of converting between domains. This is also leveraged by previous works [1], [4].

2.3. Packing and Rotation

Given the prohibitive size of a ciphertext polynomial, it is helpful to pack several integer data elements into every polynomial. With such *packing*, every (plaintext and ciphertext) $(N-1)$-degree polynomial represents an $N/2$-element vector of plaintext integer *slots* in CKKS. Packing more data into one ciphertext reduces the number of ciphers, and it yields higher throughput by performing batched computations.

However, packing limits the homomorphic functionality – for example, two elements packed within the same ciphertext cannot be added directly (note that a ciphertext has "tangled" all the plaintext coefficients). To add two elements, they must be correctly aligned to use the same slot within their respective plaintext polynomials. If they are not aligned, the polynomial must be first *rotated*. Such a rotation can be performed over ciphertexts but incurs a significant cost (discussed next). To avoid this cost, prior works like Cheetah/Gazelle [1], [2] employ a simple form of packing. While Hyena attempts more aggressive forms of packing to reap its benefits, it must also manage the overheads of any resulting rotations. Our main contribution is to devise a packing and dataflow that limits this rotation overhead.

2.4. Rotation Overheads

There are two types of allowed rotations - cyclic shifts and permutations. A shift cyclically rotates all slot values by a fixed amount and is typically achieved with a single call to Automorphism. However, there may be applications that require a non-cyclic permute of slots. This permutation can be performed by either computing a series of shifts for each slot change or by permuting coefficients within the coefficient domain. Typically, the second approach is chosen since the former requires many KeySwitchings and is expensive. Our proposals reduce the number of rotations (specifically permutes), thus also reducing the overall associated noise growth. Rotation incurs three major overheads for permutations and two for cyclic shifts, described below.

The first overhead is that the newly rotated ciphers must be multiplied with Key Switching Hints (KSH). Whenever the slots of a ciphertext are rotated, it cannot be decrypted correctly using secret key s. In fact, the ciphertext is transformed to an encryption of the plaintext but with a *different* rotated key. *Key-switching* [17] is a technique that restores a ciphertext under s' to a ciphertext under a key s. This is done by multiplying the ciphertext with a pair of polynomials called Key Switching Hints. For N possible rotations, the server has to store N KSHs. KSHs are large (e.g., 32 MB [4]), incurring a significant compute and memory cost.

The second overhead is incurred while performing permutations. If the ciphertext is in the evaluation domain (as required for fast PolyMult), it has to first be converted to the coefficient domain [2] or perform a series of many shifts. This back-and-forth $O(n \log n)$ conversion between domains makes a permutation significantly more expensive. Cyclic shifts, on the other hand, do not require such a conversion. In CKKS, Automorphism [4] is a technique that implements a cyclic shift of r places in the plaintext by applying the following shuffle σ_r of coefficient locations i in the evaluation domain: $\sigma_r : i \mapsto i \cdot r \mod N (i = 0, 1, ..., N-1)$.

The third overhead is the hardware cost of implementing the complex wiring needed by rotations. From the equation above, it is evident that even the simple shift operation requires each coefficient location to be connected to a range of coefficient locations to accommodate different r.

3. Private Inference Frameworks

Threat Model: Private inference (PI) refers to a system where a client uploads an encrypted version of their data to the cloud, and the cloud server performs the DNN classification task over the encrypted inputs. We assume a threat model where the server is honest but curious, i.e., it follows the protocol precisely but may try to infer information. Similarly, the server wants to protect the (potentially proprietary) DNN model that it executes. The PI framework must, therefore, not leak information about the DNN model to a curious client. Since the HE computations happen on the server, the weights can be processed as plaintexts to simplify computations, similar to the threat model described in [1], [2], [12], [13]. We assume the above threat model, secured with the CKKS scheme. Attacks possible here include model extraction attacks and scenarios where the server does not perform inference operations in the agreed manner. The former reveals the model hyper-parameters like the tensor shape but not the actual data. Any structured packing, similar to the large body of prior related work described in the paper, reveals

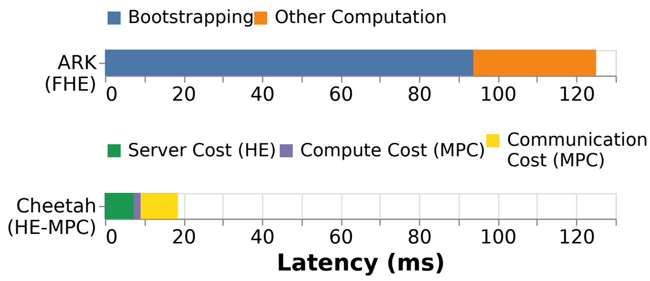


Figure 1. ResNet20 inference latency for FHE (Setup A) with ARK [35] (125 ms) and HE-MPC (Setup B) with Cheetah [1] (27 ms). Bootstrapping consumes 75% of the time. HE-MPC framework is $\sim 6.75\times$ faster. Network assumption: 2 Gbps LAN, similar to MP2ML [36].

TABLE 1. REPORTED ACCURACIES BY PRIOR HE-BASED DNNS WORKS, FOR CIFAR-10 & -100. [14] IS USED BY BTS [12] & CRATERLAKE [13].

Work	MiniONN [41]	[42]	LoLa [15]	DELPHI [5]	[14]	AESPA [43]
Setup Type	SetupB	SetupA	SetupA	SetupB	SetupA	SetupB
C10 Accuracy	81.61%	76.70%	74.10%	82.5%	92.43%	94.96%
C10 Time	544 s	6 hrs	730 s	40 s	2.9 hrs	-
C100 Accuracy	-	-	-	66%	-	77.40%
C100 Time	-	-	-	98 s	-	0.29 s

such information. Defending against the latter requires Zero-Knowledge Proof attestation, which is outside the scope of this work.

To understand the trade-off space, this paper considers multiple PI framework setups. A key differentiating factor is the handling of the DNN's non-linear activation functions (like ReLU) that are problematic for HE implementations.

Setup A (FHE): Many works [3], [14], [37], [38], [39] have approximated non-linear layers so that the *complete* classification task can be performed at the server. Performing all operations needs a boundless noise budget. This is achieved by including periodic bootstrapping operations that reset the noise. Such Fully Homomorphic Encryption (FHE) typically employs large parameter sizes (key sizes of 12-120 MB [35]) that increase the compute+memory burden. Previous works have demonstrated that bootstrapping is expensive and contributes $\sim$75% of the overall latency ($\sim$4 ms per bootstrap with a state-of-the-art accelerator [35]). Bootstrapping is, therefore, the target of recent FHE architecture papers [4], [11], [12], [13], [35], [40]. Additionally, approximate activation functions degrade the accuracy of classification. Prior works have attempted to approximate ReLU as polynomial functions but only achieved matching accuracy in small models [14], [37].

Setup B (HE-MPC): Given the drawbacks in Setup A, many studies [1], [2], [5], [7], [36], [44] have deployed frameworks that involve the client when performing non-linear functions. This Setup B solution works well as non-linear layers usually have less complexity and can be handled by the client, while the heavy work involving (potentially proprietary) weights is offloaded to the cloud. Since the client performs intermediate operations by decrypting the values, *it inherently resets the noise*. Thus, such a model needs a very small noise budget, which keeps the HE parameters small for the *same security level*, e.g., 8 KB for Hyena in Setup B vs. 56 MB when implementing Setup A [12]. Since the client resets the noise regularly, expensive bootstrapping is avoided. Additionally, unlike Setup A, there is no accuracy drop. One problem here is that model extraction attacks [45], [46] are possible if the client can observe the linear layer's input-output pairs. This is addressed by integrating HE with Multi-Party Computation (MPC) [2], [5], [36] to prevent client access to raw intermediate values while still being able to perform non-linear operations. The one major downside of Setup B is the additional back-and-forth communication with the client due to the MPC protocol. Recent works [5], [43] on HE-MPC private inference have focused on reducing this communication cost. In Figure 1, we compare the end-to-end ResNet20 inference latency for the two setups, demonstrated by their state-of-the-art works. Prior work [1] only presents server-side results for Cheetah, so we calculate the MPC costs using AESPA [43] MPC protocol by multiplying the cost of each transfer with the number of packed output features (see Section 7.4 for methodology details). We show that Setup B significantly outperforms Setup A, i.e., the cost of bootstrapping and larger HE parameters is much higher ($\sim 6.75\times$) than the cost of client-server communication.

Note that most practical MPC solutions perform a mix of MPC and non-linear layer approximation, thus affecting accuracy. The slight inaccuracy introduced by the MPC protocols is orthogonal to the techniques presented in this paper; we evaluate our ideas in the context of the state-of-the-art MPC protocol with the highest accuracy (AESPA [43]). Table 1 compares the prior HE-based DNN works in terms of their reported accuracy and inference time, which also suggests that Setup B has achieved higher accuracy and better performance across CIFAR datasets. Given the analysis in Figure 1 (calculated with AESPA) and the clear superiority of Setup B in terms of performance and accuracy, we focus our efforts on improving this setup. Our evaluations focus on the server-side latency to demonstrate the benefits of our techniques. Improving the communication protocol is orthogonal to this work - the client can use any non-linear layer protocols to interface with the server. We demonstrate 11.2 ms communication cost in baseline Setup B in Section 7.4 for ResNet-20 inference. Finally, note that while we advocate for Setup B, *Hyena techniques (packing + dataflow) are server-side optimizations and are also applicable to Setup A*, with minor changes to output feature map handling.

4. Motivation

4.1. Packing Trade-Offs

Gentry et al. [17] introduced a *packing* mechanism that allows encoding a vector of values within a ciphertext polynomial, essentially reducing the overheads of polynomials. However, this limits the functionally complete property of the HE scheme since individual values from the same ciphertext cannot be extracted and added/multiplied together. A rotation operation enables shifting/permuting the plaintext elements within these packed ciphertexts [17]. Rotation requires non-trivial data movement, and rotated ciphers must also go through Key Switching for validity under the decryption key. Combined, these steps make Rotation expensive. Overall, packing yields a smaller footprint for the polynomials but incurs a higher cost for rotations. Many prior works have leveraged packing for high efficiency [1], [2], [7], [9], [12], [13].

In the context of DNNs, there are three data structures being packed - input feature maps, weights, and partial sums. Unencrypted weights can also be fetched as scalar values, but scalar multiplication with an input feature (IF) polynomial would result in multiplication with all packed IF slots. Not all slot multiplications contribute to a valid partial sum (*psum*), wasting compute resources with ineffectual computations. Instead, weights are also encoded as polynomials in most implementations so that only valid slot multiplications happen. Moreover, fetching weights as scalars and encoding them as polynomials on the fly doesn't work either. This is because the evaluation domain conversion cost using NTT, for a faster multiplication as described in Section 2.2, ends up slowing down the critical path latency. Weights are also packed to reduce the cost of polynomials and exploit SIMD parallelism. In addition, the computations are being performed by a series of nested for-loops - the ordering of these for-loops specifies the dataflow. The dataflow dictates the reuse of operands. The combination of packing and dataflow also influences the required rotations and whether they will be shifts or permutations. To design an efficient accelerator, we must consider all of these effects. *The goal is to pack as much data as possible into each polynomial and pick a suitable dataflow (to reduce overall data movement) while keeping the rotation calls (especially permutations) to a minimum.*

We first start by characterizing state-of-the-art packing techniques: 1) LoLa packing [15], 2) Channel Packing [2], which is used by state-of-the-art FHE works [12], [13], [14], and 3) Cheetah-2PC packing [6]. We identify key bottlenecks in each of these packings and address these bottlenecks by building a basic activation+weight packing scheme. We then propose a novel application-specific packing that reduces the number of permute-type rotations (and the overall rotations) while reducing the data footprint.

4.2. LoLa Packing (LoLa-Pack)

Figure 2 shows how LoLa (Low-Latency CryptoNets) [15] packs several input feature map values in a single IF polynomial, but many copies of a *single* weight value in the W polynomial. Note that a single ciphertext PolyMult performs a parallel multiplication of corresponding elements in two plaintext vectors. A PolyMult with LoLa-Pack is able to perform a limited number of useful parallel multiplications but also several ineffectual computations, e.g., when the input feature and weight belong to different channels, as is the case for the second half of the example polynomials in Figure 2. This PolyMult generates a *packed psum* polynomial. In the next cycle, a different W polynomial is fetched and multiplied with the same IF (dataflow depicted in Figure 2). The generated *psum* cannot be readily added to the previous one since they are not aligned. Therefore, the *psum* is rotated (entailing expensive format conversion using NTT, permutation, and KeySwitching). Since the *psum* is unstructured, the rotations required to accumulate are permute-type. This is illustrated in Figure 3 where all rotations in *LoLa-Pack* are permutes. As discussed earlier, a permutation places a much larger burden on the hardware than a shift. While psums undergo frequent rotation, the key benefit of LoLa-Pack is that the IF and W poly do not require any expensive rotations.

Since a single weight value is packed in a polynomial, LoLa must fetch RSC_tK weight polynomials to reuse the same IF, resulting in a large data footprint. This is depicted in Figure 3, where LoLa-Pack reports a 20 GB data footprint. Usually, the reuse distance for weights (RSC_tK KB for 1KB polynomials) is too large for caching to be effective, and all the weight fetches end up being memory accesses. The large footprint also reflects the ineffectual computations performed in LoLa because of channel constraints and the resulting padded zeroes.

4.3. Channel Packing (Ch-Pack)

Gazelle [2] employs a packing based on single input single output (SISO) format, i.e., it uses a structured organization of input feature values that are all used to calculate one output neuron. This is shown in the top-right corner of Figure 4. It converts most permute-type rotations to cyclic shifts, which can be implemented more efficiently using Automorphism. Each polynomial tries to include elements from multiple channels, hence called *Channel Pack*. We assume an obvious added optimization where multiple SISO faces corresponding to different output neurons are also packed, enjoying the SIMD batched parallelism if slots are available after channel packing.

Channel packing rotates IFs to create new IF-W pairs, which is more expensive than rotating W [1]. Hence, the W polynomial packs a single weight from a few kernels and channels with many padded zero values. This leads to many *ineffectual computations* in the PolyMult, as well as large footprints and many rotations (as seen in Figure 3). We were able to validate the number of rotations against the equation reported by the Gazelle paper. We also note that while Ch-Pack has a very high data footprint, its reuse distance is small, which means that Ch-Pack's dataflow relies on a secondary cache to capture its data reuse. Overall, we observe 37× more data movement and 2.5× more rotations than LoLa-Pack. Recent works accelerating FHE (BTS [12], CraterLake [13], and [14]) implement Channel Packing and the corresponding dataflow (details in Figure 5).

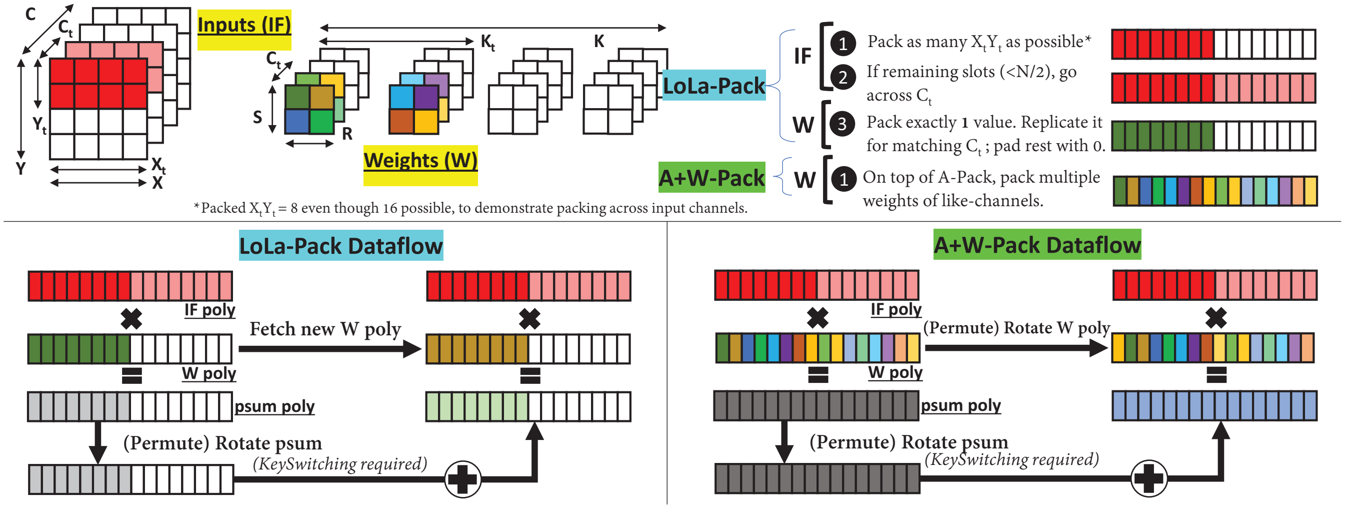

C
C$_t$
Inputs (IF)
Y$_t$
Y
X$_t$
X
C$_t$
S
R
K$_t$
K
Weights (W)
LoLa-Pack
A+W-Pack
IF
1 Pack as many X$_t$Y$_t$ as possible*
2 If remaining slots (<N/2), go across C$_t$
W
3 Pack exactly 1 value. Replicate it for matching C$_t$; pad rest with 0.
W
1 On top of A-Pack, pack multiple weights of like-channels.
*Packed X$_t$Y$_t$ = 8 even though 16 possible, to demonstrate packing across input channels.
LoLa-Pack Dataflow
IF poly
W poly
Fetch new W poly
psum poly
(Permute) Rotate psum
(KeySwitching required)
A+W-Pack Dataflow
IF poly
W poly
(Permute) Rotate W poly
psum poly
(Permute) Rotate psum
(KeySwitching required)

Figure 2. LoLa-Pack [15] and A+W-Pack for a DNN's convolutional layer. A+W packs both Activations and Weight, while LoLa only packs Activations.

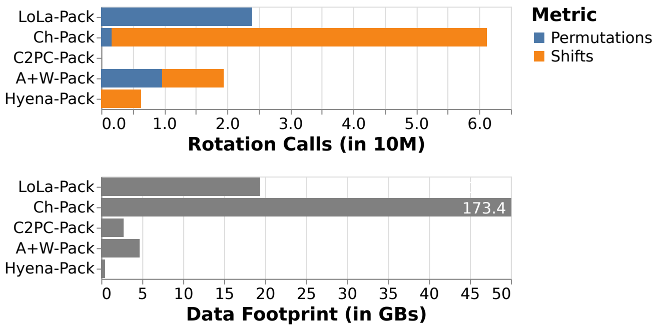


Figure 3. Packing strategies compared in terms of data footprint and rotation type for ResNet50 inference. Automorphisms (shifts) are significantly cheaper than Permutations.

4.4. Cheetah-2PC packing (C2PC-Pack)

Cheetah-2PC [6] proposes Polynomial Coefficient Encoding that packs in such a manner that the product of packed *IF* and *W* results in a *psum* which does not need intra-slot addition, and hence rotation (as seen from Figure 3). However, as noted by the authors, this only works if the combined slot utilization by *IF* and *W* is not larger than the available slots of a single cipher. The slot utilization of *IF*, *W* combined has an upper bound of nearly 50%, resulting in a sub-optimal design in terms of slot utilization. To be precise, *IF* packs inputs for M_t output neurons across C_t input channels, resulting in $M_tC_t \times RS$ packed values. A W polynomial packs RSC_t values for K_t output channels such that $M_tC_tRS + RSC_tK_t \leq N/2 + RSC_t$. We implemented Cheetah-2PC with the most optimal dataflow that tries to maximize reuse in the cache.

5. New Packing Strategies

To address the inefficiencies in prior packing approaches in LoLa-, Ch- (used in CraterLake and BTS), and C2PC-Pack, we consider new packing approaches in this section. We start with the A+W Pack, which is more aggressive in packing the weight polynomial. We then introduce the Hyena Pack, which avoids the *psum* permutations needed during aggregation.

5.1. A+W-Pack: Packing Activations and Weights

We start with more aggressive packing in the weight polynomial in addition to LoLa-pack. We do this in a manner that reduces ineffectual computations, reduces the overall data footprint, and reduces permutations.

This new *A+W-Pack* is shown in Figure 2. Given that C_t channels are packed in the *IF* ciphertext, and only like-channels must be convolved, we first start by placing the weight face (*RS*) values from the same C_t channels in the W polynomial. Since RS is usually less than X_tY_t, there might exist leftover slots in W that are filled with weight values from K_t different output channels. In short, RSC_tK_t weight values are packed in one W polynomial, greatly reducing the weight footprint reduction relative to LoLa-Pack (and quantified in Figure 3). An *IF* can be re-used over K/K_t weight polynomials, which can potentially be cached on-chip.

A+W-Pack addresses some of the under-utilization in LoLa but does not resolve the unstructured packing issues that cause permute-type rotations in *psum*. Rather, to reuse the multiple values in W, either W or *IF* must be rotated along with *psum*, increasing rotations. Yet, as seen from Figure 3, A+W-Pack decreases the total number of rotations by 19% over LoLa-Pack. This is because LoLa-Pack pads

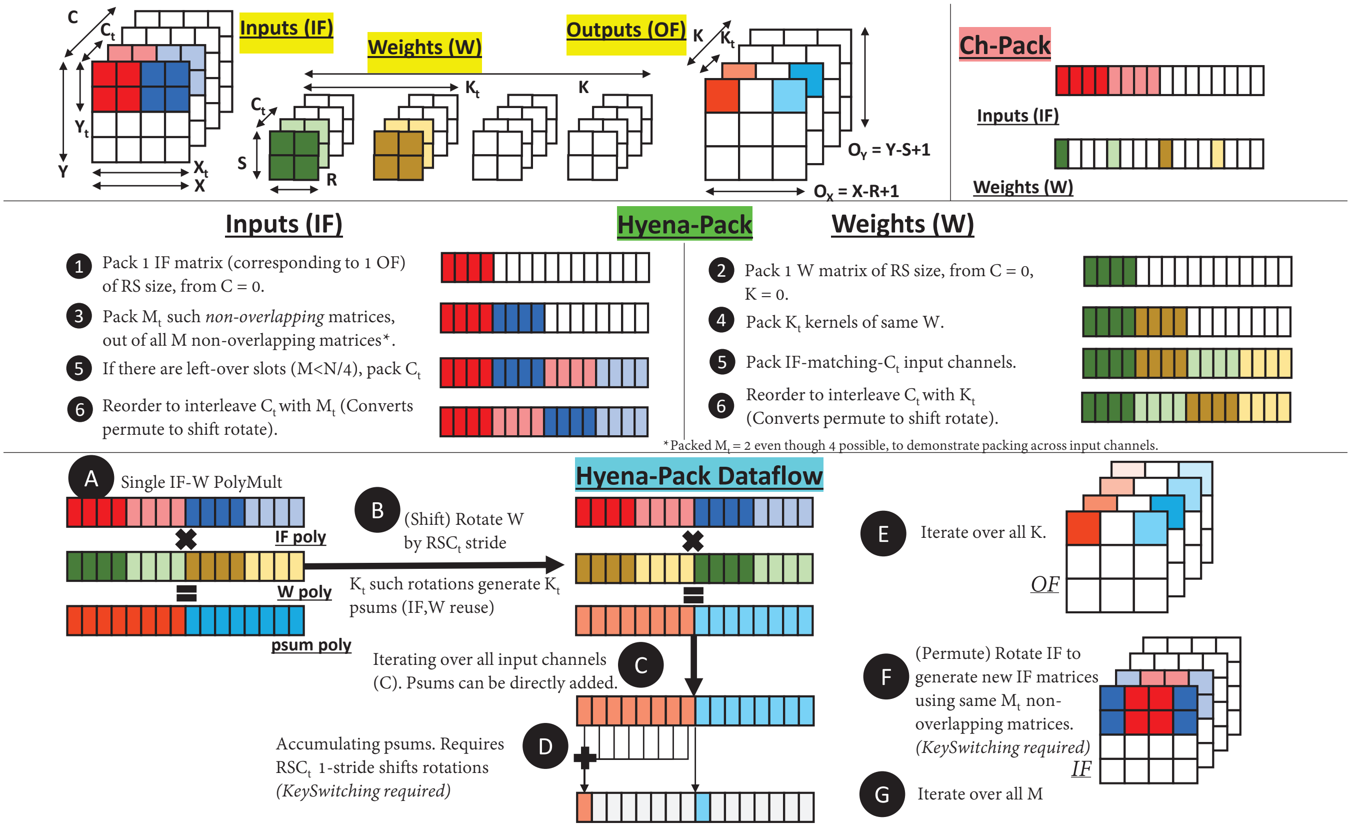

C
C_t
Y
Y_t
X_t
X
Inputs (IF)
Weights (W)
Outputs (OF)
C_t
S
R
K_t
K
K
K_t
O_Y = Y-S+1
O_X = X-R+1
Ch-Pack
Inputs (IF)
Weights (W)
Inputs (IF)
Hyena-Pack
Weights (W)
1 Pack 1 IF matrix (corresponding to 1 OF) of RS size, from C = 0.
3 Pack M_t such non-overlapping matrices, out of all M non-overlapping matrices*.
5 If there are left-over slots (M<N/4), pack C_t
6 Reorder to interleave C_t with M_t (Converts permute to shift rotate).
2 Pack 1 W matrix of RS size, from C = 0, K = 0.
4 Pack K_t kernels of same W.
5 Pack IF-matching-C_t input channels.
6 Reorder to interleave C_t with K_t (Converts permute to shift rotate).
*Packed M_t = 2 even though 4 possible, to demonstrate packing across input channels.
A Single IF-W PolyMult
IF poly
W poly
psum poly
B (Shift) Rotate W by RSC_t stride
K_t such rotations generate K_t psums (IF,W reuse)
Hyena-Pack Dataflow
C Iterating over all input channels (C). Psums can be directly added.
D Accumulating psums. Requires RSC_t 1-stride shifts rotations (KeySwitching required)
E Iterate over all K.
OF
F (Permute) Rotate IF to generate new IF matrices using same M_t non-overlapping matrices. (KeySwitching required)
IF
G Iterate over all M

Figure 4. Hyena Packing + Dataflow. Top right depicts Channel Packing [2], used in BTS [12] and CraterLake [13].

W with zeroes when $C_t > 1$, essentially performing more outer-loop PolyMults and requiring more *psum* rotations. Cheetah [1] demonstrated that it is easier to rotate W than IF since weights are in plaintext and don't need KeySwitching. This is depicted in the A+W Dataflow example shown in Figure 2.

5.2. Hyena-Pack

LoLa- and A+W-Pack require *psum* to be rotated after every IF and W multiplication in order to be accumulated. This is a more expensive task than rotating unencrypted W. Furthermore, since the *psum* values' order within each polynomial is unstructured, every rotation is permute-type, which requires complex operations. In contrast, Ch-Pack introduces a structured packing but requires every IF input to be rotated and lacks packing efficiency. Similarly, C2-PC can only achieve 50% packing efficiency. In this sub-section, the objective is to pick best practices of both types - pack IF and W in a *structured* order such that the resulting *psum* requires fewer rotations and mostly shift-type rotations in order to accumulate while retaining the high slot utilization of A+W-Pack. *The key insight is that if we align IF and W coefficients in an order that their rotated multiplications add up to the same psum coefficients, rotation of psum can be minimized.* We label this packing as *Hyena-Pack*. We supplement the packing with a dataflow that further reduces the rotation calls.

Hyena-Pack uses the following strategy, demonstrated in Figure 4. ❶ Start by packing a single RS-sized matrix of IF, of $C = 0$, corresponding to one output neuron into the IF polynomial. This *"structured"* packing is similar to single input single output (SISO) packing in Gazelle [2]. ❷ Match the weights by packing RS values (face) of $C = 0, K = 0$ in W polynomial, aligned. ❸ We then pack as many non-overlapping IF matrices[1] as possible in the IF polynomial. Avoiding overlap is helpful in reducing the IF footprint. ❹ The weight polynomial then includes more kernels from the same input channel. This increases the useful computations done by each PolyMult. It also increases the output neurons generated by each PolyMult. A side-effect is that we'll need larger on-chip storage to reuse psums. ❺ Next, we repeat this process across channels (C_t) to fill up the rest of the IF and W polynomials. Overall, IF packs $(M_tC_t \times RS)$ values and W packs $(K_tC_t \times RS)$ values. ❻ Next, we want to rotate W and reuse the IF. Since the channels have to match, we

[1] Total number of non-overlapping IF matrices are $M = \lceil \frac{X}{R} \rceil \lceil \frac{Y}{S} \rceil$, out of total number of IF matrices $T = (X-R+1)(Y-S+1)$. M_t non-overlapping matrices are placed in one IF polynomial, where $M_t = \min(\frac{N/2}{RS}, M)$.

must essentially interchange the dark weights and the light weights. The initial organization ❺ in the polynomial will require a permute. Instead, we use the organization shown in ❻. Step 🅑 in the dataflow shows that we can rotate *W* by shifting every coefficient by 8.

Hyena Pack Summary

IF polynomial organized as many non-overlapping RS-sized matrices; *W* polynomial organized such that it can be reused with simple rotations. Benefits: high packing and compute density, easy *W* shifts, and easy aggregation (discussed next).

5.3. Hyena-Dataflow

Here, we describe the Hyena dataflow that increases polynomial reuse and enables efficient aggregation. A single *IF* and *W* multiplication generates a *psum* polynomial. As seen from Step 🅐 of the dataflow in Figure 4, a single *psum* polynomial consists of several values that must be aggregated to generate M_t output neurons. This requires *psum* rotation after every *IF*-*W* multiplication. Instead, we delay the aggregation step by first rotating (plaintext) *W* in Step 🅑. Given the *structured packing*, *W* requires a RSC_t stride shift-rotate that aligns *W* with the same *IF* such that the next PolyMult generates output neurons for a different output channel. Therefore, given that K_t kernels are packed in the *W* polynomial, we can repeat Step 🅑 K_t times and generate K_t psums, giving us high reuse of both *IF* and *W* polynomials. In Step 🅒, we fetch the same *IF* and *W* corresponding to different input channels to generate psums that can be directly added to the previously generated psums (since they are correctly aligned). In Step 🅓, we finally aggregate the values co-resident in one *psum* polynomial that contributes to the same output neuron. This is implemented by performing RSC_t 1-stride shift-rotates and adding the resulting polynomials. This step can also be performed in the $\log RSC_t$ step by performing binary tree reduction, but it requires more parallel resources. Instead, we implement these steps sequentially. By delaying this accumulation from Step 🅐 to Step 🅓, we perform *psum* rotations only after input channels have been accumulated, essentially reducing the *psum* rotations. But this delay requires us to buffer several psums (about 0.5 MB) on-chip.

Once the targeted output neurons are fully generated, we calculate the same output neurons for different output channels by iterating over all *K* (Step 🅔). The reuse distance of *IF* and *W* here are $\frac{C}{C_t}$ and $\frac{CK}{C_tK_t}$, which can be cached on-chip. We further try to reuse the (possibly) cached IFs by rotating them (as shown in Step 🅕) by a stride to generate neighboring output neurons. There are $(R-1)(S-1)$ possible rotations of IFs that can generate psums for $(R-1)(S-1)$ new output neurons (some post-processing is required to handle boundary effects). These *IF* rotations are, unfortunately, permute operations. Finally, in Step 🅖, we repeat this over all *M* matrices ($\frac{M}{M_t}$ iterations). The details of the Hyena dataflow are also captured with pseudo-code in Figure 5.

Overall, Hyena-Pack retains the slot utilization efficiency of A+W-Pack by packing multiple values within both *IF* and *W*, with over 90% lower footprint than C2PC-Pack, as seen from Figure 3. Along with 67% less rotations over A+W-Pack, Hyena-Pack could convert almost all permutations to simple shifts due to its structured packing (Figure 3), which highly optimizes the implementation costs. Our structured packing+dataflow works well for both CONV and FC layers.

Hyena Dataflow Summary

The key highlights of the Hyena dataflow are as follows: (i) There are minimum ineffectual PolyMults, i.e., every step retains a high degree of packing in *IF*, *W*, and *psum* polynomials. (ii) *IF* and *W* polynomials are reused with simple shifts to *W*, then psums are reused by iterating over input channels. (iii) After significant *psum* reuse and aggregation, psums are further aggregated with simple shifts. (iv) An outer loop performs an expensive permute operation to slide the *IF* polynomial. (v) 7 MB of buffering is required to capture reuse in *IF*, *W*, and *psum* polynomials.

6. Implementation Details

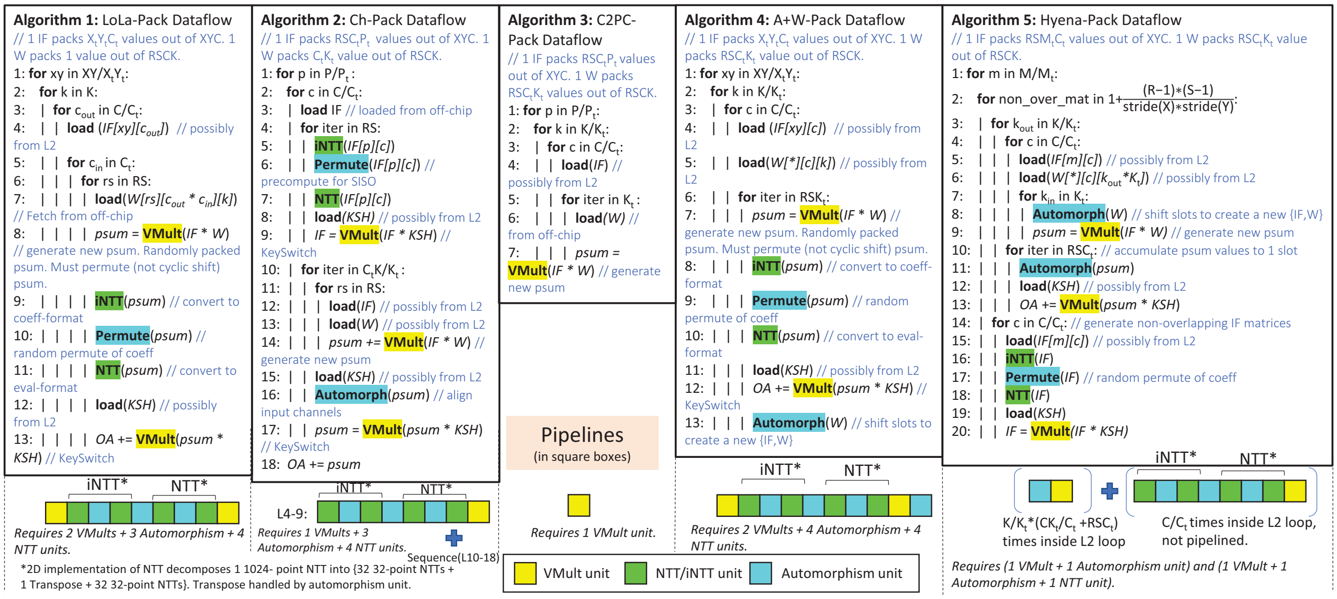
6.1. Architecture Overview

We next describe the hardware accelerator we use to evaluate the Hyena packing and dataflow. The overall organization of the chip design is shown in Figure 6. The chip is composed of a High Bandwidth Memory (HBM) interface and several tiles that include an L2 and a processing element (PE). Each L2 has a capacity of 14 MB allocated across *IF*, *W*, and KSH polynomials. Each PE is designed to handle a polynomial of degree 1K and 32-bit coefficients. Larger degree polynomials utilize multiple PEs. A range of HE parameters are supported by breaking down larger bit-widths using RNS. Each PE consists of a set of registers (for *IF*, *W*, and *psums*), modular multiply-accumulates (MACs), NTT, and Rotation units. The Hyena architecture is designed by borrowing insights from the state-of-the-art FHE architectures [4], [12], [13], but for smaller HE parameters.

6.2. Dataflow

Figure 5 and Table 2 are a detailed enumeration of pseudo-code that implements each of the dataflows discussed in Sections 4.1 & 5. The code handles a single layer of a DNN. Given a specific packing scheme, each dataflow is designed

[2] *P* denotes number of Output neurons per channel. *Non* refers to overlapping IF matrices (See Section 5.2). Note: degree of packing ($*_t$) depends on respective packings.



```
Algorithm 1: LoLa-Pack Dataflow
// 1 IF packs $X_tY_tC_t$ values out of XYC. 1 W packs 1 value out of RSCK.
1: for xy in $XY/X_tY_t$:
2:   for k in K:
3:   | for $c_{out}$ in $C/C_t$:
4:   | | load ($IF[xy][c_{out}]$)  // possibly from L2
5:   | | for $c_{in}$ in $C_t$:
6:   | | | for rs in RS:
7:   | | | | load($W[rs][c_{out} * c_{in}][k]$) // Fetch from off-chip
8:   | | | | $psum$ = VMult($IF * W$) // generate new psum. Randomly packed psum. Must permute (not cyclic shift) psum.
9:   | | | | iNTT($psum$) // convert to coeff-format
10:  | | | | Permute($psum$) // random permute of coeff
11:  | | | | NTT($psum$) // convert to eval-format
12:  | | | | load($KSH$) // possibly from L2
13:  | | | | $OA$ += VMult($psum * KSH$) // KeySwitch
```

```
Algorithm 2: Ch-Pack Dataflow
// 1 IF packs $RSC_tP_t$ values out of XYC. 1 W packs $C_tK_t$ value out of RSCK.
1: for p in $P/P_t$ :
2:   for c in $C/C_t$:
3:   | load IF // loaded from off-chip
4:   | for iter in RS:
5:   | | iNTT($IF[p][c]$)
6:   | | Permute($IF[p][c]$) // precompute for SISO
7:   | | NTT($IF[p][c]$)
8:   | | load($KSH$) // possibly from L2
9:   | | $IF$ = VMult($IF * KSH$) // KeySwitch
10:  | for iter in $C_tK/K_t$ :
11:  | | for rs in RS:
12:  | | | load($IF$) // possibly from L2
13:  | | | load($W$) // possibly from L2
14:  | | | $psum$ += VMult($IF * W$) // generate new psum
15:  | | load($KSH$) // possibly from L2
16:  | | Automorph($psum$) // align input channels
17:  | | $psum$ = VMult($psum * KSH$) // KeySwitch
18:  $OA$ += $psum$
```

```
Algorithm 3: C2PC-Pack Dataflow
// 1 IF packs $RSC_tP_t$ values out of XYC. 1 W packs $RSC_tK_t$ values out of RSCK.
1: for p in $P/P_t$:
2:   for k in $K/K_t$:
3:   | for c in $C/C_t$:
4:   | | load($IF$) // possibly from L2
5:   | | for iter in $K_t$ :
6:   | | | load($W$) // from off-chip
7:   | | | $psum$ = VMult($IF * W$) // generate new psum
```

```
Algorithm 4: A+W-Pack Dataflow
// 1 IF packs $X_tY_tC_t$ values out of XYC. 1 W packs $RSC_tK_t$ value out of RSCK.
1: for xy in $XY/X_tY_t$:
2:   for k in $K/K_t$:
3:   | for c in $C/C_t$:
4:   | | load ($IF[xy][c]$)  // possibly from L2
5:   | | load($W[*][c][k]$) // possibly from L2
6:   | | for iter in $RSK_t$:
7:   | | | $psum$ = VMult($IF * W$) // generate new psum. Randomly packed psum. Must permute (not cyclic shift) psum.
8:   | | | iNTT($psum$) // convert to coeff-format
9:   | | | Permute($psum$) // random permute of coeff
10:  | | | NTT($psum$) // convert to eval-format
11:  | | | load($KSH$) // possibly from L2
12:  | | | $OA$ += VMult($psum * KSH$) // KeySwitch
13:  | | | Automorph($W$) // shift slots to create a new {IF,W}
```

```
Algorithm 5: Hyena-Pack Dataflow
// 1 IF packs $RSM_tC_t$ values out of XYC. 1 W packs $RSC_tK_t$ value out of RSCK.
1: for m in $M/M_t$:
2:   for non_over_mat in $1+\frac{(R-1)*(S-1)}{stride(X)*stride(Y)}$:
3:   | for $k_{out}$ in $K/K_t$:
4:   | | for c in $C/C_t$:
5:   | | | load($IF[m][c]$) // possibly from L2
6:   | | | load($W[*][c][k_{out}*K_t]$) // possibly from L2
7:   | | | for $k_{in}$ in $K_t$:
8:   | | | | Automorph($W$) // shift slots to create a new {IF,W}
9:   | | | | $psum$ = VMult($IF * W$) // generate new psum
10:  | | for iter in $RSC_t$: // accumulate psum values to 1 slot
11:  | | | Automorph($psum$)
12:  | | | load($KSH$) // possibly from L2
13:  | | | $OA$ += VMult($psum * KSH$)
14:  | for c in $C/C_t$: // generate non-overlapping IF matrices
15:  | | load($IF[m][c]$) // possibly from L2
16:  | | iNTT($IF$)
17:  | | Permute($IF$) // random permute of coeff
18:  | | NTT($IF$)
19:  | | load($KSH$)
20:  | | $IF$ = VMult($IF * KSH$)
```

iNTT*
NTT*
$K/K_t*(CK_t/C_t + RSC_t)$ times inside L2 loop
C/C_t times inside L2 loop, not pipelined.
Requires (1 VMult + 1 Automorphism unit) and (1 VMult + 1 Automorphism + 1 NTT unit).
🟨 VMult unit
🟩 NTT/iNTT unit
🟦 Automorphism unit

*2D implementation of NTT decomposes 1 1024- point NTT into {32 32-point NTTs + 1 Transpose + 32 32-point NTTs}. Transpose handled by automorphism unit.

Figure 5. Dataflows and pipeline associated with packing schemes. *In order*: LoLa-Pack [1], Ch-Pack [2], [12], [13], C2PC-Pack [6], A+W-Pack, Hyena-Pack. The code to model these has been released as part of the HEPack-Sim tool described in Section 7.1.2.

TABLE 2. OPERATION COUNT AND DATA MOVEMENT DISTRIBUTION FOR VARIOUS PACKING SCHEMES.

Packing	**Operations**		**Off Chip Data Movement**	
LoLa	ROT	$\frac{XY}{X_tY_t} \times RSCK$	IF	$\frac{XY}{X_tY_t} \times \frac{CK}{C_t}$
	MUL	$\frac{XY}{X_tY_t} \times RSCK$	WT	$\frac{XY}{X_tY_t} \times RSCK$
	NTT	$\frac{XY}{X_tY_t} \times RSCK$	PSUM	$\frac{XY}{X_tY_t} \times (\frac{2CK}{C_t} + RSCK)$
Ch	ROT	$\frac{P^2}{P_t} \times (\frac{RSC}{C_t} + \frac{CK}{K_t})$	IF	$\frac{P}{P_t} \times \frac{C}{C_t}$
	MUL	$\frac{P}{P_t} \times \frac{RSCK}{K_t}$	WT	$\frac{P}{P_t} \times \frac{RSCK}{K_t}$
	NTT	$\frac{P}{P_t} \times \frac{C}{C_t} \times RS$	PSUM	$\frac{P}{P_t} \times \frac{CK}{C_tK_t}$
C2PC	ROT	0	IF	$\frac{P}{P_t} \times \frac{CK}{C_tK_t}$
	MUL	$\frac{P}{P_t} \times \frac{CK}{C_t}$	WT	$\frac{P}{P_t} \times \frac{CK}{C_t}$
	NTT	0	PSUM	$\frac{P}{P_t} \times \frac{K}{K_t}$
A+W	ROT	$\frac{XY}{X_tY_t} \times \frac{2RSCK}{C_t}$	IF	$\frac{XY}{X_tY_t} \times \frac{CK}{C_tK_t}$
	MUL	$\frac{XY}{X_tY_t} \times \frac{RSCK}{C_t}$	WT	$\frac{XY}{X_tY_t} \times \frac{CK}{C_tK_t}$
	NTT	$\frac{XY}{X_tY_t} \times \frac{RSCK}{C_t}$	PSUM	$\frac{XY}{X_tY_t} \times \frac{K}{K_t}$
Hyena	ROT	$\frac{M \times Non}{M_t}(\frac{CK}{C_t} + RSC_tK/K_t + C/C_t)$	IF	$\frac{M}{M_t}$
	MUL	$\frac{M \times Non}{M_t} \times \frac{CK}{C_t}$	WT	$\frac{M}{M_t}$
	NTT	$\frac{M \times Non}{M_t} \times \frac{C}{C_t}$	PSUM	$\frac{M \times Non}{M_t} \times \frac{K}{K_t}$

to reuse polynomials and reduce rotation overheads. Each dataflow also results in a different execution pipeline on the accelerator. Table 2 counts the number of times each operation is called and the resulting data movements per data structure. This result is a property of each packing scheme, while Figure 5 highlights the lines of code that exercise the vector-multiply unit, the NTT/iNTT unit, and the Automorphism unit. For instance, any cipher rotation would also call a sub-routine Key-Switch, which is not labeled in the table but is represented by the functional units used on the chip. Below each dataflow, we list the sequence in which these units are invoked, which helps construct a balanced hardware pipeline for each dataflow.

Each dataflow is designed to create new combinations of IF and W inputs. The five dataflows also differ in ways that impact the architecture: (a) Whether to rotate W or IF or fetch newer versions: A+W-Pack and Hyena-Pack rotate their packed weights, and Ch-Pack rotates IF, but require an additional Automorphism/Permute over W/IF to prepare for the next iteration. LoLa- and C2PC-Pack avoid Automorphism by fetching a new unpacked weight every step, most likely from off-chip (this latency can be hidden by fast HBM2 for LoLa). (b) Partial sum accumulation: LoLa-Pack and A+W-Pack have unstructured packing, and hence the psums that contribute to a single output neuron are scattered across coefficients. Generating an output neuron requires permutation, which is expensive because it invokes the NTT unit. Previous works [4], [12], [13] have employed Ch-Pack. Hyena-Pack also uses a structured packing that avoids permutations during *psum* aggregation. Both Ch-Pack and Hyena-Pack are able to do this because they aggregate psums across input channels. C2PC does not require any rotations. (c) The loop orders are designed specifically to maximize the reuse of expensive ciphertexts, with priority $psum > IF > W$. In all five algorithms, RS face values and then input channels are navigated for a full output neuron.

6.3. Functional Units

We next describe each hardware unit in Hyena.

VMULT, VADD unit: HE operations are modular arithmetic and need modular multiplications and additions. We assume a vector of modular multiply/add units (VMULT/-VADD) within the PE. F1 [4] proposed an optimized modular

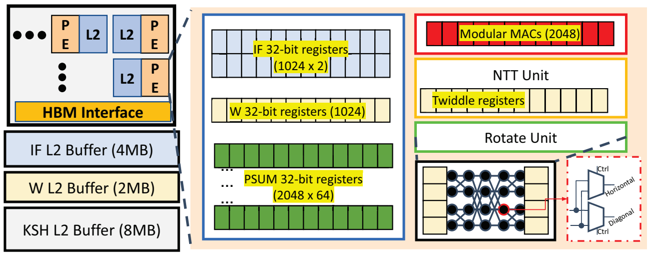


Figure 6. Hyena Architecture.

multiplier based on the observation that for a restricted set of bit-widths (q), a specialized low area/power multiplier can be designed. Since both Hyena and F1 use 32-bit coefficients, we utilize their multiplier in our modular MAC unit. A vector MAC unit performs polynomial multiplications for both $IF \times W$ and $* \times KSH$ (for Key Switching operation).

Registers: We dedicate registers for storing one W polynomial, a pair of IF polynomials, and 64 pairs of *psum* polynomials. A+W-Pack and Hyena-Pack pack along output channels (till $K_t = 64$) for reusing IF and W. Hence, Hyena-Arch uniquely dedicates 64 *psum* registers to avoid misses during accumulation. Similar to CraterLake [13], our FUs are chained, and the input of one FU is passed to another through internal latches.

L2 Buffers: A+W-Pack, Ch-Pack, and Hyena-Pack demonstrate reuse beyond registers. Hyena-Pack reuses *psum* within registers by iterating over all input channels (Step C of Figure 4), which requires different IF and W. We capture the re-use distance of these polynomials (C/C_t for IF, KC/K_tC_t for W) by caching up to 512 IF and 4K W polynomials in an L2 buffer per PE. KSHs are large (for N possible cipher rotations, we need N KSH polynomials) and are often required during cipher rotation. Therefore, KSHs are fed from a dedicated KSH buffer.

Automorphism unit: Rotation is expensive because Automorphism and permutation dictate that any register storing a coefficient value can go to any other register of the same polynomial. A crossbar provides the non-blocking behavior where this permutation can happen in $O(1)$, but it does not scale well (complexity increases by N^2).

Recent proposals F1 [4], BTS [12], and CraterLake [13] are designed for FHE (Setup A), where parameters have to be larger. Their rotation units are, therefore, overkill for Hyena, which uses Setup B and can deploy HE with smaller parameters. Given the smaller value for N, we observe that a Benes network [47], [48], [49] is enough to efficiently implement permutation within registers, as depicted in Figure 6. Benes is a non-blocking N-input N-output multi-stage network with $2\log(N)+1$ levels, each with N 2×2 switches. The non-blocking property and latch-free stage implementation allow the Benes network to communicate with any destination without contention and in $O(1)$ delay, making permutations (or transpose) on N elements $O(1)$. Furthermore, the cost of $O(N\log N)$ switches is small, in terms of area and power, for a small $N = 1K$.

NTT unit: NTT converts polynomials by performing $\log N$ recursive steps of N modular multiplications with constant multiplicative values (twiddles). Given N MACs, NTT requires $\log N$ cycles. We adopt F1's implementation that decomposes NTT into multiple smaller degree NTTs [4], [50]. Its transpose operation is performed using the Benes network, and its butterfly is performed using wide vector mults. This unit also consists of a small register file that stores twiddles.

6.4. Pipelining of Functional Units

While past works like F1 have developed general-purpose architectures to support a range of FHE algorithms, our focus is on PI, which allows us to tailor the architecture and resources to cater to the needs of the most effective dataflows. Note that the architecture is still flexible enough to execute a range of dataflows and workloads. In particular, we observe that the Hyena dataflow has a specific pipeline of operations and units that are invoked (see bottom of Figure 5) - we allocate resources to each of these units so that the pipeline is balanced and under-utilization is kept to a minimum.

As seen in Figure 5, the Hyena dataflow reads input operands from its L2 buffers, followed by calls to VMult, VAdd, and Rotation units in its inner loops (lines 4-13 of Algorithm 5). With a Benes network used for rotation, our cycle time is typically determined by VMult. Each PE implements 2 VMults (note that each cipher consists of 2 polynomials), each with 1K 32-bit multipliers, 1 1K-input Benes network, 32 32-point NTT units, and registers as described in Section 6.3. All these functional units are fully pipelined, so they achieve the same throughput (1K 32-bit elements/cycle) and are chained so that the input of one unit is passed to the next without the need to read/write $IF/psum/W$ registers. In addition, as part of the outer for loop, we must periodically perform permutes (Lines 14-20) that slide the IF window and prepare for future PolyMults. As we'll discuss next, those have a simpler pipeline that executes off the critical path.

The bottom of Figure 5 elaborates on the pipeline stages needed for each of the different dataflows on Hyena-Arch. Due to the simplicity of LoLa-, C2PC-, and A+W-Pack, it is straightforward to design a pipelined execution for all IF-W pairs, as shown. However, Ch-Pack involves multiple loops that cannot be fused because of dependencies. While Lines 4-9 in Algorithm 2 can be executed in a pipelined fashion, the next set of operations (Lines 10-18) require many sequential calls to VMults, resulting in longer latency. Because this latter phase is dominant, it cannot be pipelined with the operations in Lines 4-9. This sequential latency is an impediment to its performance.

The Hyena dataflow includes a similar step (Lines 14-20 of Algorithm 5) that computes new IF combinations. However, the Hyena dataflow is designed such that this permutation step is independent of earlier *psum* computation and can, therefore, be performed in parallel with the execution of Lines 3-13, i.e., this permutation is not on the critical path as long as we dedicate enough NTT/Automorphism units.

TABLE 3. Hardware Requirements of different packing on Hyena-Arch.

	LoLa	Ch	C2PC	A+W	Hyena
Number of VMults, Automorphs, NTTs	(2,3,4)	(1,3,4)	(1,0,0)	(2,4,4)	(2,2,1)
VMult Area (mm^2)	7	3	3	7	7
Automorph Area (mm^2)	18	18	0	25	12
NTT Area (mm^2)	26	26	0	26	6
Registers and L2 Area (mm^2)	114				136
Total Area (mm^2)	**167**	**162**	**117**	**173**	**163**

TABLE 4. Hyena Architecture Parameters for Hyena-Pack

Hyena-Arch PE at 200MHz, 163.02 mm^2				
Component	**Params**	**Spec**	**Power (W)**	**Area (mm^2)**
MAC unit	count vector length	2x2 256	0.494	7.326
NTT (with Twiddles)	count vector length	2x32 32	0.968	6.743
Rotate (Benes)	count, cycles	2x2, 1	2.062	12.65
IF Registers	count	2K	0.008	3.1635
W Registers	count	1K	0.004	1.5817
PSUM Registers	count	2Kx64	0.552	25.308
IF L2	size	2MB	3.801	31.466
W L2	size	4MB	2.793	15.751
KSH L2	size	8MB	6.066	59.023
Total			**16.75 W**	**163.02 mm^2**

We do not require a long hardware chain of these units and we are able to meet the deadline by reusing a few of these units.

Based on the above pipelines, we create different instances of the Hyena architecture, where resource allocations are such that they optimize the pipeline for each packing/dataflow. The corresponding chip area requirements are summarized in Table 3, along with per-component breakdowns. We see that the Hyena dataflow requires modest hardware that is smaller than the hardware needed by LoLa-Pack and A+W-Pack. C2PC-Pack needs the least hardware because it only needs a VMult unit. Hyena-Pack has higher area requirements as it needs a few NTT/Automorphism units that can be reused.

> **Hyena Architecture Summary**
>
> The use of small HE parameters enables efficient rotation with a simple Benes network. Each pack/dataflow requires a different series of invocations (Figure 5) of the key hardware units. A custom architecture for each dataflow implements varying chained resources to create a balanced pipeline, notably many *psum* registers (Table 3). The Hyena pack/dataflow is amenable to an architecture that has low area requirements and low latency per inference.

7. Evaluation

7.1. Methodology

7.1.1. Modeling Hyena and Baselines. To model Hyena and prior works, we develop a simulator, HEPack-Sim, that models the different aspects of packing, dataflow, and architecture. We describe HEPack-Sim in Section 7.1.2. We were able to validate several metrics for our baselines against data in prior works. The benchmarks selected for the evaluation include 2 CNNs (ResNet50 [51] and MobileNetV2 [52]) with ImageNet dataset and a Neural machine translation model (GNMT [53]). We report the latency and energy results summed across all their layers, including the CONV and FC layers. Hyena architecture is combined with various pack+dataflows: LoLa-pack [15], Ch-Pack from Gazelle [2], C2-PC from Cheetah-2PC [6], A+W-Pack, and Hyena-Pack. LoLa-Pack can exploit weight sparsity and is evaluated with 50% sparsity. The number of functional units is tailored to suit the HE parameters and to match the throughput of 1024 elements/cycle for every stage. Each dataflow lends itself to a different pipeline (Section 6.4), which impacts the resource counts for each component.

We model the architecture components in Verilog, implemented using industry-standard synthesis (Synopsis DC) and place-and-route (Synopsis IC Compiler) tools in a 65 nm CMOS process at 200 MHz frequency. The energy, area, and access latencies of L2 SRAM caches are derived from Cacti 6.5 at 65 nm technology. We assume 3.9 pJ/bit energy for HBM2 accesses [54]. Table 4 summarizes all Hyena architecture parameters. While our work applies to most HE schemes, we evaluate a popular scheme, CKKS [22], to obtain parameters $(n, \log Q) = (1024, 32)$ for depth-1 with 128-bit security that fit within a single PE. As mentioned in Section 3, the hybrid HE-MPC protocol allows us to keep the noise budget small and, hence, the HE parameters small. Our inference model (similar to [1], [2]) can work with small depth since output neurons after every layer are sent to the user, who resets the noise.

7.1.2. HEPack-Sim. We build a discrete-event functional simulator, HEPack-Sim, to benchmark and validate our proposal and compare its benefits to the existing research. Specifically designed for evaluating Homomorphic Encryption-based Machine Learning inference tasks on an accelerator, this simulator is configured with the parameters of accelerator hardware, Homomorphic Encryption (HE) scheme, and the neural network (NN) layers. It then calculates the number of execution cycles and accesses to each hardware component to complete one NN input.

HEPack-Sim operates by handling the different layers of the NN one at a time. It starts by calculating the packing parameters based on the shape of the layer. It then triggers specific events like VMult, NTT, and Automorphism based on layer operations and employs a batching strategy to group multiple layer operations. Each event unfolds as a series of underlying operations; for example, a VMult event involves reading a vector of *IF* and *W* registers, performing vector modulo multiplication, and writing the result to PSUM registers. The simulator keeps track of all operations happening based on the events executed, allowing us to analyze the results on all levels - individual operations, specific events or layers, or the entire NN run as a whole.

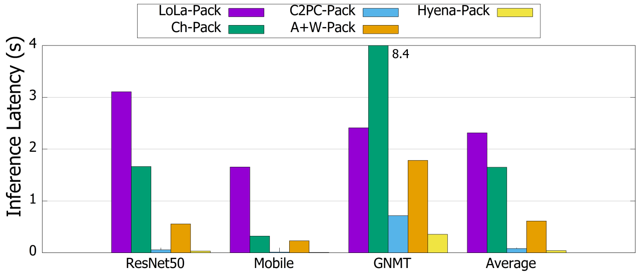


Figure 7. Performance comparison of various packing.

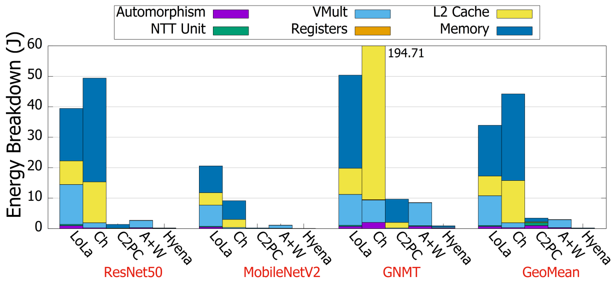


Figure 8. Per-component energy breakdown for various packing.

As described in Section 6.2, each packing has its data flow (and an associated pipeline), which is implemented in the simulator (similar to what is presented in Figure 5). The simulator keeps track of access to the register, multiplier, and other hardware resources, which are used for estimating the final energy costs. Because the accelerator implementation is fully pipelined, each pipeline execution increments the cycle count, with the slowest functional unit determining the cycle time. For most packing schemes, regular and sequential memory requests are prefetched at a rate that is lower than the available memory bandwidth, i.e., we are not memory bandwidth bound. For example, LoLa-Pack has a high memory bandwidth requirement but fetches a new weight polynomial only once every 45 ns. As discussed later in Section 7.2, C2PC-Pack is memory bound; a polynomial is requested every 5 ns, so the compute throughput is determined by the rate at which polynomials can be transferred across the memory channel. Note that if HE parameters are scaled, other packing schemes may also end up being memory-bound and will have a performance estimation similar to how C2PC-Pack is modeled. The simulator assumes energy and latency numbers for each hardware unit, estimated separately with the methodology described in Section 7.1.1. HEPack-Sim is available open-source for reproducibility and for exploring novel techniques to optimize private inference at https://github.com/UtahArch/HEPack-Sim.

7.2. Impact of Packing

Figures 7 & 8 show server-side latency and energy for a range of packing schemes. A+W-Pack addresses the slot under-utilization in LoLa-Pack and reduces the footprint. The 228× reduction in memory accesses plays a large role in overall energy savings of 11×. LoLa also pads weights with zeroes, generating ineffectual computations and, hence, 3.7× slowdown over A+W-Pack. Note that Cheetah [1] reports 200 ms ResNet50 inference latency for LoLa-Pack, but their chip is 3.6× larger, runs at twice the frequency, and is modeled at 5 nm technology.

Ch-Pack introduces structured packing but suffers from poor slot utilization (on average, only 28% of the slots are filled). While Ch-Pack on ResNet50 performs 1.9× faster than LoLa-Pack, we observe 9× higher data movement in Ch-Pack than LoLa, contributing to 1.3× higher energy. GALA [55] adapts Ch-Pack by delaying the *psum* rotation required to aggregate across packed input channels until all psums across related ciphers are added. This dataflow reduces the *psum* rotations by $\frac{C}{C_t}\times$ but requires all IFs corresponding to that *psum*. This trade-off is not accelerator-friendly, as IFs can no longer be tiled and served from an on-chip cache. Hence, while GALA reduces rotations by 10× over Ch-Pack, it results in 1.5× higher energy consumption.

C2PC-Pack [6] avoids rotations altogether by using a mathematical trick that aggregates psums across slots during multiplication itself, which results in a superior design over prior packings. However, as seen from Section 4.4, this trick restricts the combined slot utilization of IF and W to be a maximum of near 50%. Typically, this prioritizes IFs over W, resulting in low slot utilization of W, leading to higher memory activity. We noticed that since almost every PolyMult needs a new W poly and the W reuse distance is too large to be cached, the performance is bottlenecked by how fast the memory can provide W polys. Meanwhile, other packings reuse W using rotations, offsetting the cost of memory fetch. Also, C2PC is unlikely to scale well with HE parameters due to larger sizes.

Hyena-Pack's dense and structured packing, complemented with its dataflow, avoids cipher rotation in the innermost loop - *psum* is rotated after aggregation over C/C_t channels (Figure 4). By saving rotation and data movement, Hyena-Pack outperforms Ch-Pack (A+W-Pack) with 38× (14×) speedup and 162× (11×) lower energy. Hyena-Pack is 1.9× faster than C2PC-Pack with Hyena-Arch. This is because C2PC-Pack requires a new IF-W pair for every multiplication, resulting in always being memory bound (low Op/byte ratio), and hence, we observe no effect due to architecture. Similarly, due to its poor data movement and cipher reuse, C2PC-Pack consumes 5.3× more energy than Hyena-Pack. Given that C2PC-Pack is memory-bound, future accelerator innovations will have no effect on its performance. Overall, assuming the 7 ms *unencrypted* inference latency target mentioned in the Google TPU analysis [32], Hyena is ∼5× slower for ResNet50. Server-side latencies for private inference using Hyena are 32.8 ms, 8.8 ms, and 356.5 ms for ResNet50, MobileNetV2, and GNMT, respectively.

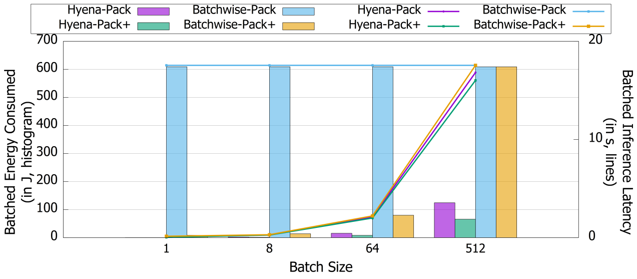


Figure 9. Effect of Batch Size on Hyena, compared to BatchWise-Pack's Batch-axis packing [7]. Both axes depict results for batched inference on ResNet50.

7.3. Batching

All five packing schemes, LoLa, Ch, C2PC, A+W, and Hyena-Pack, support batched inputs by sequentially executing each input, given fixed resources. An alternate technique, proposed by CryptoNets [3] and used by nGraph-HE2 [7] and [56], avoids rotations altogether by performing an *IF* batch-wise packing. Similar query-wise batched packing is also used in the context of HE-based COVID-19 strain classification [57], [58] using NN. Essentially, a single *IF* value across multiple batches is packed in an *IF* and multiplied with a single weight value replicated for batch size in W. This allows multiple inputs to be processed simultaneously - increasing throughput without sacrificing total batched latency in some cases. Moreover, as the next set of IF/W remains aligned, psums can be accumulated without rotations. We compare the performance and energy of this batched scheme, termed *BatchWise-Pack* in Figure 9, for batch-size of 1, 8, 64, and 512 for the ResNet50 benchmark. We use a tiled dataflow that reduces the memory fetches by caching IF, W at steps of RSC_t in L2. As the batch size increases, we see that the latency for Hyena-Pack increases linearly since Hyena processes batches sequentially while BatchWise-Pack remains the same till batch 512. As a result, it outperforms Hyena at the batch of 512. It is important to note that a larger batch size would also result in linear time/energy increment for BatchWise-Pack as there are no slots available in the cipher to parallelize. Furthermore, Figure 9 represents the total batched latency, a metric of throughput, and not the per-batch latency. If we update Figure 9 to show response times instead, Hyena would be 512x better than BatchWise packing (for a batch size of 512) since Hyena schedules per-batch inference. BatchWise-Pack's long response times may not be palatable in real-world latency-sensitive deployments.

We make an interesting observation from BatchWise- and Hyena-Pack that there are two degrees of SIMD parallelism within packing that do not require neighbor slot communication - batch and output channel. Hyena was designed at a batch size of 1 to pack multiple channels and hence significantly outperforms BatchWise at low batches, which only packs one output channel. Therefore, we first modify BatchWise to pack output channels if slots are available and label it as *BatchWise-Pack+*. As seen from Figure 9, it achieves high efficiency at low batches as well. In fact, it performs better than Hyena in layers with low output channels and low batch size because Hyena has internal rotations for each output channel. Therefore, we came up with a Hyena packing scheme (termed *Hyena-Pack+*) that includes batches - prioritize packing across batches over input channels. Hyena+ is faster than Hyena and BatchWise+, even at large batch sizes. Overall, at a batch size of 512, Hyena+ is 1.09× faster and consumes 9.25× lower energy than BatchWise-Pack+.

TABLE 5. Comparison of works targeting ResNet20 Private Inference.

	[14]	F1+ [4]	Crater-Lake [13]	BTS [12]	ARK [35]	**Hyena**
Time	2.9hrs	2.69s	249.4ms	1.91s	125ms	HE=0.2ms MPC=11.2ms
Tech	on Intel Xeon 8280	12/14nm	12/14nm	7nm	7nm	65nm
Area (mm^2)		636	472.3	373.6	418.3	163.02
Power		-	317W	133.8W	281.3W	16.75W
On-chip storage		256MB	256MB	512MB	512MB	15.02MB

Figure 9 also described an important trade-off study - security v/s performance. As described in the threat model, model hyper-parameters are revealed in Hyena-Pack (this is true for all prior works that attempt structured packings). An alternative way is to pack across batches (BatchWise-Pack), but such a packing performs poorly at small batch sizes. Note that small batch size is typical in most inference applications since large batch sizes incur much higher response latencies. The performance increases significantly when the packing scheme is made aware of the hyper-parameters, as seen by the Batch-Pack+ scheme.

7.4. FHE v/s Hybrid HE-MPC

As discussed in Section 3, the entire DNN can be executed with FHE on the server (Setup A). Bootstrapping is the major overhead in this setup, requiring high data movement and compute (hundreds of Giga integer ops and hundreds of GB memory accesses with parameters (N=64K, L=60, logq=28), resulting in slow execution (>300 ms/bootstrap on a NVIDIA Tesla V100 [11]). Moreover, to reduce the number of bootstrapping calls, FHE typically employs large parameters (56 MBs per cipher [12]). Most of this paper focuses on Setup B, where we involve the client in performing non-linear layers using MPC. Our hybrid HE-MPC [1], [2], [5], [7], [36] allows us to reduce our parameters (8 kB ciphers) for the same security level and avoid bootstrapping, but requires client-server interaction communication. We use a state-of-the-art MPC solution for an accuracy preserving low degree polynomial activation function [43] for non-linear layers in DNNs and consider LAN of 2 Gbps [36], [59]. We compare the inference latency of ResNet20 of Hyena with prior works in Table 5, with their architecture costs. All numbers are taken directly from the papers. F1+ is an implementation of F1 [4] with a larger area demonstrated by [13] on ResNet20. We make the following conclusions: (a) Due to small parameter requirements, Hyena can achieve orders of magnitude higher performance. (b) The communication cost observed accounts for 99% of the overall inference latency after Hyena's techniques to reduce server cost but is still lower than performing FHE. (c) Due to low compute requirements, Hyena needs simpler hardware and is the smallest chip with lower power and on-chip storage requirements. (d) Most previous works use FHE-friendly implementations of ResNet20 on MNIST/CIFAR datasets [14]; larger DNN models are unlikely to scale well with FHE in terms of both efficiency and accuracy. (e) Finally, all Hyena techniques are server-side optimizations and thus are applicable to FHE workflow as well, with small post-processing changes after each layer.

8. Related Works

Since its innovation in 2009 by Gentry [16], researchers have rushed to make Homomorphic Encryption practical. Encrypted neural network inference algorithms were first proposed in CryptoNets [3], which demonstrated 5 orders of magnitude slower inference as compared to unencrypted inference over CPU. In recent years, the improvements over CryptoNets can be summarized in two categories: 1) Techniques for basic computations, such as convolutions [15], matrix-multiplications [60], and non-linear functions [14], [38], [61]; 2) Combining different cryptographic primitives (HE, garbled circuits, oblivious transfer) to achieve better performance for the evaluations of linear and (approximating) non-linear functions in the inference process. Below are some efforts that narrow the gap between mathematical formulation and practical application.

Combining HE with other cryptographic primitives: Due to the massive compute burden posed by Fully Homomorphic Encryption, many researchers have explored combining HE with another cryptographic primitive to trade off compute costs with communication/security. For instance, Gazelle [2], DELPHI [5], CrypTFlow2 [62], and Cheetah [1] use HE to evaluate the linear functions of DNN and turn to garbled circuits or oblivious transfer to compute the nonlinear functions of the inference process. Such hybrid HE-MPC is a widely accepted design choice but comes at the cost of high communication, as reported by CHOCO [44]. Recent works [5], [43] on HE-MPC PI have focused on reducing this communication cost. In this work, we build a fast and practical inference framework based on two-party computation and HE and demonstrate low communication latency. Some other works consider the secure inference problem with more than two parties, mainly using secure multiparty computation, such as [63], [64], [65].

HE acceleration: Due to the complexity of homomorphic computation, relying on software optimizations alone cannot achieve desirable results. Hence, in the past several years, numerous works have studied hardware acceleration of homomorphic encryption in the scenarios of encrypted inference. Many works have attempted to utilize the high compute parallelism offered by GPUs [11], [66], [67], [68]. GPUs are typically bottlenecked by their limited on-chip storage capacity [68]. Jung et al. [11] propose HE kernel fusion to reduce this bottleneck and show practical bootstrapping. FPGA designs have also been gaining attention for HE [40], [69], [70], but still remain an order of magnitude slower than the state-of-the-art ASICs. Recently, HE has seen a growing interest in the architecture community, resulting in ASIC designs [1], [4], [12], [13], [35]. Cheetah [1] is the first HE-based ML work and targets a hybrid HE-MPC design, reporting ResNet50 inference latency of 200 ms with a 587 mm^2 chip dissipating 30 W in 5 nm. F1 [4] was the first ASIC to target CKKS bootstrapping but only performed unpacked bootstrapping. CraterLake [13] and BTS [12] both optimize for packed bootstrapping. ARK [35] is the state-of-the-art FHE ASIC accelerator that eliminates off-chip data movement bottleneck through runtime data generation and inter-operation key reuse, resulting in private ResNet20 inference latency of 125 ms, with a 418 mm^2 chip that has 512 MB of on-chip storage and consumes 281 W power. Instead, Hyena employs hybrid HE-MPC, which keeps the parameters and hence the on-chip requirement low (15.02 MB), resulting in a low area (163 mm^2) and low power (16.75 W) design.

Packing techniques: The high cost of HE can be amortized by batching a set of computations in the slots offered by the CKKS encoding scheme. However, as discussed in Section 4, one must be very careful while packing as intra-slot computations are expensive, requiring complex procedures and large keys. We discuss many such packing schemes in detail in Section 4. Early works like CryptoNets [3] focused on using slots for SIMD parallelism by packing the same pixel across different batches in the same cipher. Recently, more nuanced packing schemes have come up that pack pixels and kernels such that the number of slot rotations is minimized. One such packing, Channel-Packing [2] has been widely adopted by many works, like CraterLake [13], BTS [12], [14]. We demonstrate that Channel-Pack suffers from poor slot utilization and propose Hyena-Pack, which offers an order of magnitude improvement in terms of the number of rotations and data movement, translating directly to performance and energy improvement. A recent packing work [71] (used by ARK [35]) noted that a high stride CONV layer introduces gaps in the output pixel packing, introducing a cascading slot under-utilization issue. They address this with a multiplexed parallel convolution technique. However, Hyena advocates for a hybrid HE-MPC setup, where such gaps are filled during the re-encryption process done after every layer by the client.

9. Conclusion

While prior works have performed a limited amount of packing to keep rotation overheads in check, we show that with an appropriate packing and dataflow strategy, dense packing is practical. We devise a packing and dataflow that leads to compact data footprints, high reuse, and few permutations. This is augmented with an architecture where the pipeline and resources are tailored for the dataflow.

Compared to a range of packing/dataflow baselines, we show significant improvements in latency and energy, e.g., 38× speedup and 162× better energy than widely used Channel packing. We study the trade-offs between FHE and hybrid HE+MPC approaches and make a case for the latter for practical privacy-preserving inference. As a result, we demonstrate end-to-end ResNet20 inference latency of 11.4 *ms*, using a 163 mm^2 accelerator dissipating 16.75 *W*. While we present a novel packing for Machine Learning applications, the general principles we introduce to reduce rotations can be used to extend Hyena to other applications.

Acknowledgments

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grants CCF 2224463, CCF 2217154, and Intel.

References

[1] B. Reagen, W.-S. Choi, Y. Ko, V. Lee, H.-H. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference," in *27th International Conference on High Performance Computer Architecture (HPCA)*, 2021.

[2] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *{USENIX} Security Symposium*, 2018.

[3] 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*, 2016.

[4] 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 *54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[5] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[6] Z. Huang, W.-j. Lu, C. Hong, and J. Ding, "Cheetah: Lean and Fast Secure {Two-Party} Deep Neural Network Inference," in *USENIX Security Symposium (USENIX Security 22)*, 2022.

[7] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019.

[8] S. S. Roy, A. C. Mert, Aikata, S. Kwon, Y. Shin, and D. Yoo, "Accelerator for Computing on Encrypted Data," Cryptology ePrint Archive, Report 2021/1555, 2021, https://ia.cr/2021/1555.

[9] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does Fully Homomorphic Encryption Need Compute Acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

[10] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020.

[11] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021.

[12] 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 *ISCA*, 2022.

[13] 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.

[14] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, 2022.

[15] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *International Conference on Machine Learning*. PMLR, 2019.

[16] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009.

[17] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2012.

[18] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Annual Cryptology Conference*, 2012.

[19] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*, 2013.

[20] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," in *Cryptology ePrint Archive Report 2012/144*, 2012.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, 2014.

[22] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*, 2017.

[23] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2021.

[24] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2015.

[25] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, 2020.

[26] "Encrypted Computing Palisade Library - Duality Technologies," https://dualitytech.com/palisade/, accessed: 2021-07-10.

[27] CryptoExperts, "FV-NFLlib," 2016, https://github.com/CryptoExperts/FV-NFLlib/.

[28] M. Research, "Simple Encrypted Arithmetic Library (SEAL)," 2016, https://github.com/microsoft/SEAL/.

[29] V. Lyubashevsky, C. Peikert, and O. Regev, "On Ideal Lattices and Learning with Errors Over Rings," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2010, pp. 1–23.

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

[31] A. Sanyal, M. Kusner, A. Gascon, and V. Kanade, "TAPAS: Tricks to accelerate (encrypted) prediction as a service," in *International Conference on Machine Learning*, 2018.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke,

A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of ISCA-44*, 2017.

[33] H. L. Garner, "The residue number system," in *Western joint computer conference*, 1959.

[34] R. T. Moenck, "Practical fast polynomial multiplication," in *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, 1976.

[35] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse," in *IEEE/ACM (MICRO)*, 2022.

[36] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: a mixed-protocol machine learning framework for private inference," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020.

[37] R. E. Ali, J. So, and A. S. Avestimehr, "On polynomial approximations for privacy-preserving and verifiable relu networks," *arXiv preprint arXiv:2011.05530*, 2020.

[38] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Annual International Cryptology Conference*, 2018.

[39] K. Mihara, R. Yamaguchi, M. Mitsuishi, and Y. Maruyama, "Neural Network Training With Homomorphic Encryption," *arXiv preprint arXiv:2012.13552*, 2020.

[40] 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.

[41] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[42] 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.

[43] J. Park, M. J. Kim, W. Jung, and J. H. Ahn, "AESPA: Accuracy preserving low-degree polynomial activation for fast private inference," *arXiv preprint arXiv:2201.06699*, 2022.

[44] M. van der Hagen and B. Lucia, "Client-optimized algorithms and acceleration for encrypted compute offloading," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[45] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in *USENIX Security*, 2016.

[46] R. N. Reith, T. Schneider, and O. Tkachenko, "Efficiently stealing your machine learning models," in *Proceedings of the 18th ACM workshop on privacy in the electronic society*, 2019.

[47] S. Arora, F. T. Leighton, and B. M. Maggs, "On-line algorithms for path selection in a nonblocking network," *SIAM Journal on Computing*, 1996.

[48] A. Chakrabarty, M. Collier, and S. Mukhopadhyay, "Matrix-based nonblocking routing algorithm for Beneš networks," in *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE, 2009.

[49] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training," in *Proceeding of HPCA*, 2020.

[50] D. H. Bailey, "FFTs in external or hierarchical memory," *The journal of Supercomputing*, 1990.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[52] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

[53] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[54] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. Keckler, and W. Dally, "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems," in *Proceedings of MICRO*, 2017.

[55] Q. Zhang, C. Xin, and H. Wu, "GALA: Greedy computation for linear algebra in privacy-preserved neural networks," *arXiv preprint arXiv:2105.01827*, 2021.

[56] 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*, 2020.

[57] A. Akavia, B. Galili, H. Shaul, M. Weiss, and Z. Yakhini, "Efficient privacy-preserving viral strain classification via k-mer signatures and FHE," *Cryptology ePrint Archive*, 2023.

[58] J. Zhou, B. Lei, and H. Lang, "Homomorphic multi-label classification of virus strains," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2022.

[59] "IEEE SA Standards Board Approvals," 09/10 February 2021, https://standards.ieee.org/about/sasb/sba/feb2021/.

[60] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018.

[61] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *Annual International Cryptology Conference*. Springer, 2020.

[62] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[63] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[64] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "{SWIFT}: Superfast and Robust {Privacy-Preserving} Machine Learning," in *USENIX Security Symposium*, 2021.

[65] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *Symposium on Security and Privacy (SP)*. IEEE, 2020.

[66] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[67] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Transactions on Emerging Topics in Computing*, 2019.

[68] S. Kim, W. Jung, J. Park, and J. H. Ahn, “Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus,” in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020.

[69] S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, “FPGA-based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data,” in *25th International Conference on High Performance Computer Architecture (HPCA)*, 2019.

[70] 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.

[71] 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.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper introduces Hyena, a new framework for encrypted inference. The paper identifies three critical performance bottlenecks in encrypted inference systems: packing, reuse, rotations. Hyena overcomes them with new co-designed packing and dataflow techniques that can achieve more dense packing, less rotations, and better reuse. The results show that Hyena provides significant speedups and improved energy efficiency compared to several baselines.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

A.3. Reasons for Acceptance

1) The paper provides a comprehensive analysis of performance bottlenecks in encrypted inference systems, pinpointing the key contributors. It then introduces new techniques that better align with how data is processed and computed on, showing that dense packing is practical and offers significant performance improvements over state of the art approaches.
2) The authors publicly release HEPack-Sim for independent analysis and reproducibility by other researchers.