RTPU: Unifying Non-Private and Private Inference with Reconfigurable Architecture

Fuping Li^{1,3,4}, Ying Wang$^{2,3(\boxtimes)}$, Yinghao Yang^{1,3}, Jingxuan Li^{2,3},
Yibo Du^{2,3}, Huawei Li^{1,3}, Yinhe Han^{2,3}, Hang Lu$^{1,3,4(\boxtimes)}$, Xiaowei Li$^{1,3,4(\boxtimes)}$
SKLP, Institute of Computing Technology, CAS^1
CICS, Institute of Computing Technology, CAS^2
University of Chinese Academy of Sciences^3, *Zhongguancun Laboratory*^{4}

***Abstract*—With the rise of fully homomorphic encryption-based private inference, data centers are anticipated to simultaneously handle two disparate computational demands: plaintext-based non-private inference (NPI) and ciphertext-based private inference (PI). Unfortunately, current solutions face challenges in addressing this trend. They either depend on costly, inflexible dedicated accelerators or utilize general-purpose hardware with inferior performance. This limitation underscores the urgent need for a unified architecture capable of serving both normal and privacy-sensitive users with high efficiency.**

However, the fundamental disparities in computation patterns and resource management between NPI and PI make their architectural fusion intricate. To bridge this gap, we explore their inherent similarities and apply fine-grained reconfiguration to maximize resource sharing. We propose RTPU, a reconfigurable multi-core architecture that can seamlessly switch between tensor-based plaintext and polynomial ring-based ciphertext computations. Building upon its reconfigurable computing fabric and parallelization mechanism, we introduce a kernel group-based scheduling strategy to optimize hardware utilization and QoS. Experimental results show that: i) The RTPU architecture achieves near-ASIC performance and beyond-ASIC flexibility with substantial silicon reuse between NPI and PI. ii) The RTPU scheduler sustains high resource utilization for multi-tenant workloads with varying privacy requirements.

I. Introduction

Recent years have witnessed the widespread deployment of cloud-based neural network (NN) inference services in various applications, including image recognition, language processing, and recommendation [1]. As illustrated in Fig. 1(a), modern data centers increasingly serve users with divergent privacy requirements. Normal users send their data in plaintext format to servers for *non-private inference (NPI)*. In contrast, privacy-sensitive users who do not trust the data center require *private inference (PI)* techniques [2] to secure their data. One of the most promising solutions is *fully homomorphic encryption (FHE)*, often regarded as the holy grail of cryptography [3]. FHE allows cloud servers to perform inference on encrypted data without decryption and has been adopted by several cloud service providers. For instance, Amazon integrates FHE in SageMaker endpoints for secure real-time inference [4], and IBM provides beta-stage privacy-preserving computation services [5]. *Thus, there is a predictable trend of coexisting NPI and PI in data centers.*

However, as highlighted in Fig. 1(b), existing paradigms building upon dedicated accelerators and general-purpose architectures fall short of efficiently addressing this emerging trend. Although developing distinct accelerators for NN and FHE workloads, such as TPU [6] and SHARP [7], achieves significant inference latency reduction, this strategy suffers from doubled non-recurring engineering (NRE) costs associated with design, verification, and mask set fabrication. These performance-hungry accelerators typically require advanced process nodes like 7 nm [6], [7], which can incur additional expenses exceeding hundreds of millions of dollars [8]. Furthermore, the architectural divergence between NN and FHE accelerators leads to

Hang Lu, Ying Wang, and Xiaowei Li are corresponding authors. This work was partially supported by the National Key R&D Program of China (Grant No. 2023YFB4404400) and the National Natural Science Foundation of China (Grant No. 62172387, 62222411).

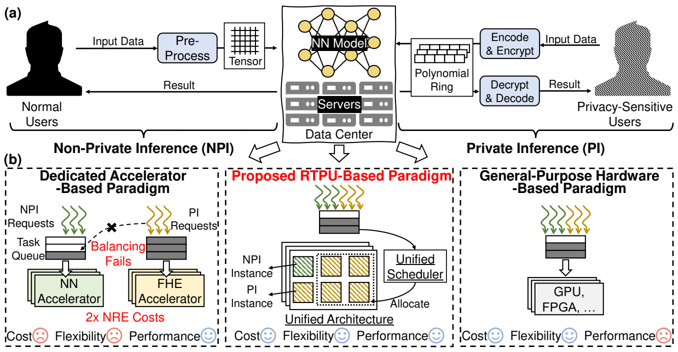

Fig. 1. (a) Overview of cloud-based NPI and PI. (b) Comparison of three accelerator design paradigms.

poor flexibility, hindering the utilization of idle hardware resources during imbalanced NPI and PI loads. As a result, data centers are compelled to deploy redundant accelerators to ensure QoS, which exacerbates hardware acquisition costs. Alternative solutions implemented on highly programmable architectures including GPUs [9] and FPGAs [10] can provide low NRE costs and superior flexibility. However, these platforms are far from ideal due to their compromised performance and increased power consumption. The above analysis raises a critical research question: *How can we design a unified architecture that simultaneously satisfies the demands of NPI and PI while achieving ASIC-like performance?*

Unfortunately, two major obstacles hinder the development of high-performance unified accelerators for NPI and PI. The first challenge stems from the fundamentally different algebraic structures underlying plaintext-based NPI and ciphertext-based PI. The former relies on *tensor* operations [6], whereas the latter utilizes *polynomial ring* algebras [11]. These distinct structures involve incompatible data widths and computational rules, making it difficult to effectively reuse hardware resources between NPI and PI. The second challenge concerns resource allocation for multi-tenant inference services with varying privacy requirements. Although supporting such services is crucial for optimizing hardware utilization and QoS, existing runtime scheduling strategies like PREMA [12] and Planaria [13] are inadequate in the context of mixed privacy-level tasks. Specifically designed for tensor-based computations, these strategies fail to capture the unique characteristics of polynomial ring-based operations. Furthermore, their simplistic parallelization approaches significantly exacerbate hardware underutilization within large-scale systems handling both NN and FHE tasks.

Our goal is to tackle these challenges and design a versatile platform that efficiently serves both normal and privacy-sensitive users. As shown in Fig. 1(b), we propose the *Ring&Tensor Processing Unit (RTPU)* architecture, which leverages the internal connections between the NPI and PI kernels to accelerate both plaintext-based and ciphertext-based inference. We also introduce an architecture-dependent scheduling mechanism to fully harness the resources for mixed NPI/PI workloads. *Compared to prior approaches using dedicated accelerators and general-purpose architectures, the RTPU-based paradigm exhibits comprehensive advantages in cost, flexibility, and performance.* In summary, our key contributions include:

- We propose a novel Ring&Tensor Core (RTC) design to accelerate the computation and data layout transformation kernels of NPI and PI. It can be configured into various modes at runtime, facilitating resource sharing between NPI and PI.
- To handle the divergent resource demands of NPI and PI kernels, we propose a multi-core RTPU architecture. It incorporates both intra-kernel and inter-kernel parallelization mechanisms to reduce latency and enhance resource utilization.
- To accommodate the reconfigurable computing fabric and parallelization mechanism of the RTPU, we propose a kernel group-based scheduler for elastic resource allocation that accounts for the heterogeneity and QoS constraints of NPI and PI tasks.
- Evaluation results demonstrate that the RTPU architecture maintains 97.43% silicon reuse across NPI and PI domains. It only incurs minor performance overhead for individual NPI/PI tasks compared to dedicated TPU/SHARP, and achieves considerable speedups for mixed tasks: 1.44× over heterogeneous accelerator platform and 20.62× over general-purpose GPU. Furthermore, the RTPU scheduler delivers average latency improvements of 3.27× and 1.79× compared to PREMA and Planaria, respectively. We also conduct design space exploration to identify optimal parameters that balance NPI and PI efficiencies.

II. Background and Motivation

A. Landscape of NPI and PI Acceleration

The ubiquitous NPI has spurred the proliferation of NN accelerators [14], which typically incorporate numerous systolic arrays or vector units to efficiently perform tensor operations such as general matrix multiplication (GEMM). Beyond boosting performance, an important lesson learned is the necessity of supporting multi-tenancy [6]. In cloud-based NN deployments, multiple inference tasks with distinct network structures, data precisions, and QoS requirements, frequently share a single accelerator. To maximize resource utilization, contemporary multi-tenant schedulers like PREMA [12] and Planaria [13] partition hardware resources along time and space dimensions, scheduling tasks using latency estimation models. However, these solutions are constrained to NN-specific accelerators with limited computing resources and do not efficiently support large-scale systems that handle both NN and FHE workloads.

The FHE-based PI is known for its robust privacy guarantee as well as significant computational overhead [15]. To mitigate the resulting slowdown, highly programmable GPUs and FPGAs [9], [10] are explored as immediate solutions, despite their inferior performance and substantial power consumption. ASIC-based accelerators [7], [11] are then proposed to provide optimized performance, effectively narrowing the gap between plaintext-based and ciphertext-based inference. However, ASICs face challenges such as high NRE costs and lengthy design cycles. Moreover, the lack of reconfigurability makes them difficult to adapt to various scenarios. Recent efforts to enhance ASIC flexibility focus on parameter variations [16], algorithmic diversity [17], and performance scalability [18], [19]. Nonetheless, these works remain confined to the PI domain.

B. Hierarchical Description of NPI and PI

As shown in Fig. 2, the software stacks of NPI and PI are hierarchically organized into three layers: model layer, operation layer and kernel layer. Users utilize high-level descriptions to define

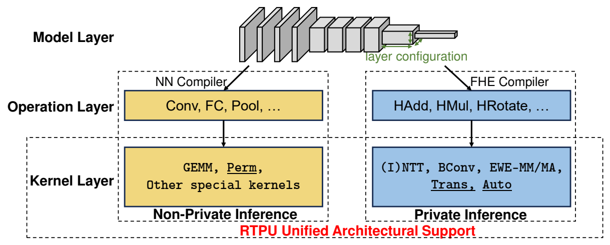

Fig. 2. Hierarchical software stacks of NPI and PI. The underlined kernels are for data layout transformation, and the others are for computation.

the NN layer configurations, such as sizes of input feature maps. Leveraging NN and FHE compilers, the model is initially transformed into a sequence of operations and subsequently lowered into multiple kernels, whose semantics align with hardware units.

1) Kernels of NPI: The operands of NPI kernels are tensors, i.e. multi-dimensional arrays. `GEMM` plays a crucial role since most linear NN operations can be converted into matrix-matrix multiplications, accounting for over 70% of the total runtime [20]. Non-linear operations, such as ReLU, cannot be represented as `GEMM` and are viewed as special kernels. The permutation kernel (`Perm`) that reorders tensor dimensions is frequently used in tensor reshaping [21].

2) Kernels of PI: In this paper, we focus on the CKKS scheme [22] due to its prevalent adoption in PI. Notably, other schemes like BGV and BFV share plenty of kernels with it [23]. At the operation layer, CKKS operates on ciphertexts created by encoding and encrypting messages that contain multiple real or complex numbers. Each ciphertext $[\![\mathbf{m}]\!]$ is composed of two polynomials, $A_{\mathbf{m}}(X)$ and $B_{\mathbf{m}}(X)$, which are defined over the polynomial ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$ with coefficients modulo Q and degree N. And $A_{\mathbf{m}}(X) = \sum_{i=0}^{N-1} A_i x^i$. The supported homomorphic operations include addition (HAdd), multiplication (HMult), and rotation (HRotate). Complex procedures like key-switching and bootstrapping can be broken down into sequences of these operations and are thus excluded from the software stack. At the kernel layer, CKKS adopts the residue number system (RNS) representation to circumvent expensive large-number arithmetic. It splits a polynomial $a(X)$ into multiple smaller *limbs* $\{[a(X)]_{q_i}\}$ where $q_i \ll Q$. Consequently, the operands of PI kernels are polynomial matrices with limbs as row vectors. The computation kernels are summarized as follows:

`(I)NTT`: The (inverse) number theoretic transform kernel converts polynomials between coefficient-based and point value-based representations. By leveraging NTT, the multiplication of polynomials $a(X)$ and $b(X)$ is performed as $a(X)\cdot b(X) = \mathtt{INTT}(\mathtt{NTT}(a(X)) \odot \mathtt{NTT}(b(X)))$, where the operator $\odot$ represents the element-wise multiplication of their coefficients. This approach reduces the complexity of polynomial multiplication from $O(N^2)$ to $O(N\log N)$. The transformation of $\hat{a}(X) = \mathtt{NTT}(a(X))$ is formally defined as:

$$\hat{a}_j = \sum_{i=0}^{N-1} a_i \cdot \omega^{ij} \bmod q, j \in [0, N-1] \tag{1}$$

where a_i and $\hat{a}_j$ correspond to the i-th and j-th coefficients of polynomials $a(X)$ and $\hat{a}(X)$, respectively. ω represents the twiddle factor. An N-point `NTT` is conventionally implemented via a butterfly network with $\log_2 N$ stages, each containing $N/2$ butterfly units [24]. However, this structure becomes prohibitive for large N. To address this, current implementations typically adopt 2D (or four-step) NTT that decomposes a large kernel into smaller ones with sizes N_I and N_J such that $N = N_I \cdot N_J$ [25]. This approach can be extended to higher dimensions, e.g. 3D NTT, where $N = N_I \cdot N_J \cdot N_K$ [23]. Additional kernels like matrix transposition (`Trans`) are inserted between the execution of `(I)NTT` kernels with different sizes.

`BConv`: The basis conversion kernel expands or shrinks the limbs of an RNS-based polynomial, which is essential for key-switching. As shown in Eq. (2), polynomial $a(X)$ with the modulus set $\mathcal{C} = \{q_i\}$ is converted to $\hat{a}(X)$ with another set $\mathcal{B} = \{p_j\}$ [11]. Here $\hat{q}_i = \prod_{k \neq i} q_k$. Notably, the second step $(\cdot \hat{q}_i \bmod p_j)$ dominates the computation time, corresponding to modular multiplication between a $|\mathcal{B}| \times |\mathcal{C}|$ base table matrix and a $|\mathcal{C}| \times N$ polynomial matrix [11].

$$[\hat{a}(X)]_{\mathcal{B}} = \left\{ \sum_{i=0}^{|\mathcal{C}|-1} \left([a(X)]_{q_i} \cdot \hat{q}_i^{-1} \bmod q_i \right) \right. \\ \left. \cdot\, \hat{q}_i \bmod p_j \right\}, j \in [0, |\mathcal{B}| - 1] \tag{2}$$

`EWE-MM/MA`: The element-wise kernel performs modular multiplication or addition between corresponding coefficients of two polynomials, or between the coefficients of a polynomial and a scalar.

Besides the aforementioned matrix transposition (`Trans`), automorphism (`Auto`) is crucial for polynomial data reordering. When a ciphertext is rotated by an amount r, it maps the i-th coefficient of the polynomial to $\phi_r(i)$-th position, where $\phi_r(i) = i \cdot 5^r \bmod N$.

3) Kernel Parallelization Strategies: To enhance computational efficiency, two complementary parallelization strategies can be used. *Intra-Kernel parallelization* divides a single kernel into multiple subkernels for parallel execution, applicable to various dimensions of input tensors and polynomials for NPI and PI [11], [26]. *Inter-Kernel parallelization* allows for the concurrent execution of multiple kernels with data dependencies through fine-grained pipelines. For instance, splitting large batches into smaller microbatches enables the overlapping of multiple consecutive `GEMM` kernels [26].

C. Motivation of Efficient Architectural Fusion

Existing general-purpose hardware exhibits an inferior architectural fusion of NPI and PI. A representative example lies in modern GPUs: While Tensor Cores significantly boost NPI workloads, executing PI kernels requires algorithmic modifications to utilize these components, which inevitably increase time complexity [9]. To address these limitations, we design RTPU to provide native support for both NPI and PI kernels, as shown in Fig. 2. This approach is driven by three key insights: **Insight 1 (I1)**: The computation and data layout transformation kernels in NPI and PI follow similar dataflows. For example, `BConv` defined in Eq. (2) primarily involves modular multiplication between the base table matrices and the polynomial matrices [11], akin to `GEMM` in NPI. This similarity allows for significant sharing of interconnects among processing elements (PEs). These interconnects occupy a substantial portion of chip area [11]. **Insight 2 (I2)**: The high-bitwidth modular arithmetic units used in PI can be decomposed into a series of low-bitwidth integer units that are compatible with NPI. **Insight 3 (I3)**: Both NPI and PI accelerators are dominated by a large amount of on-chip memory, facilitating silicon reuse across tasks with different privacy requirements. As a result, our RTPU efficiently fuses NPI and PI, maximizing the reuse of interconnects, compute units, and memory resources.

III. RTPU Architecture and Scheduler

A. Overview

We propose RTPU, a versatile system designed to perform both plaintext-based NPI for normal users and ciphertext-based PI for privacy-sensitive users. The RTPU *architecture* (see §III-B) comprises three key components: i) a multi-mode RTC accelerating diverse NPI and PI kernels, ii) a hybrid parallelization mechanism reducing kernel latencies and enhancing hardware utilization, and iii) memory and NoC designs optimized for NPI/PI data access and communication patterns. Guided by a unified performance model, the kernel group-based RTPU *scheduler* (see §III-C) operates in two stages: i) an offline optimization stage generating multi-version configurations, and ii) an online scheduling stage dynamically allocating resources and selecting configurations for all tasks.

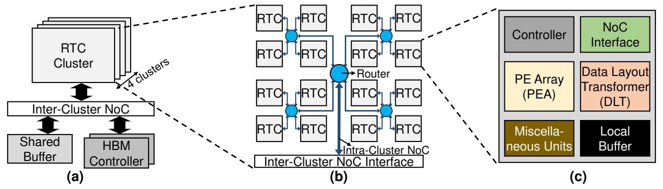

Fig. 3. (a) Overview of the RTPU architecture. (b) Organization of the RTC cluster. (c) Organization of the RTC.

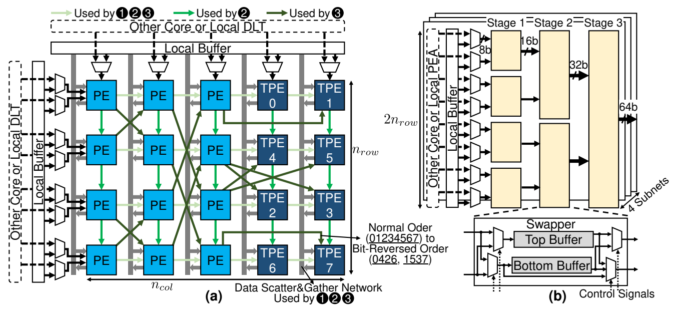

Fig. 4. (a) PEA structure, with ❶, ❷, and ❸ representing SIMD, systolic array, and butterfly network dataflows, respectively. (b) DLT structure. Here we use $n_{row} = 4$ as an example.

B. Hierarchical RTPU Architecture

Fig. 3 illustrates the proposed RTPU architecture, which consists of RTC clusters, a shared buffer, and HBM controllers. These components are hierarchically interconnected through intra-cluster and inter-cluster network-on-chips (NoCs). The subsequent sections elaborate on the RTC design, RTPU-specific kernel parallelization mechanism, and on-chip memory and NoC constructions.

1) Single Core Design: As depicted in Fig. 3(c), an RTC comprises a controller, an NoC interface, a PE array (PEA), a data layout transformer (DLT), and a local buffer. Additionally, the miscellaneous units encompass components dedicated to NPI such as activation units [21], and those dedicated to PI like PRNG generators [11]. Both the PEA and DLT are highly configurable, playing crucial roles in efficiently executing NPI and PI kernels.

PEA: As illustrated in Fig. 4(a), the 2D PEA, consisting of n_{row} rows and n_{col} columns, is designed to handle *all essential computation kernels* detailed in §II-B, namely `GEMM`, `(I)NTT`, `BConv`, and `EWE-MM/MA`. Notably, existing reconfigurable computation units [27], [28] provide only partial support for these kernels. *The proposed inter-PE topology efficiently merges three demanded inter-PE dataflows: SIMD, systolic array, and butterfly network (**I1**).* To cater to the SIMD-friendly dataflow for `EWE`, the PEA can be divided into n_{col} SIMD units chained via horizontal links, with each unit containing n_{row} lanes. This allows multiple `EWE` kernels to be mapped onto the array and executed in a pipelined manner. To further satisfy the high bandwidth requirement of SIMD, we utilize a data scatter&gather network capable of unicasting n_{row} operands to or collecting n_{row} results from PEs in each column per cycle. Additionally, horizontal and vertical links are used to support orthogonal data transmissions in systolic arrays [21] for `GEMM` and `BConv`. These kernels can also leverage the data scatter&gather

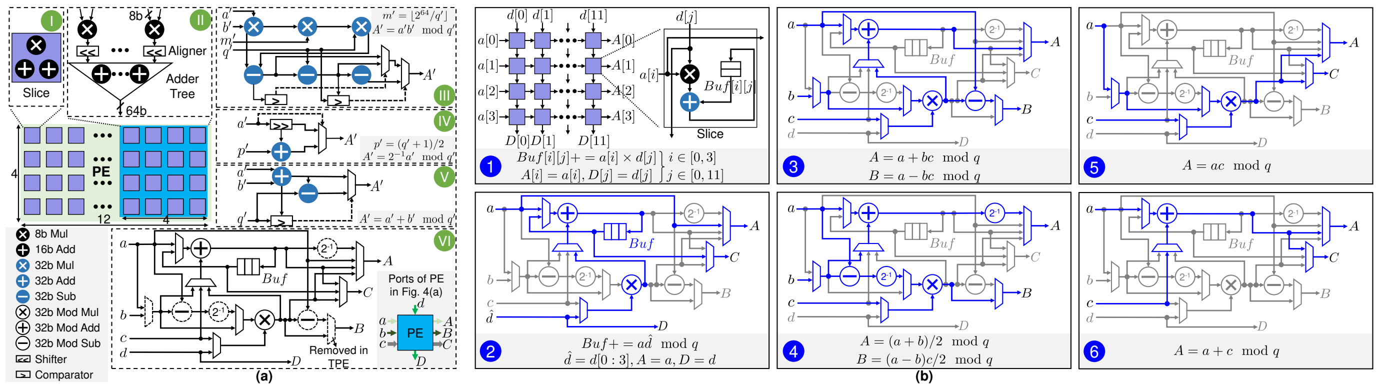

Fig. 5. (a) Organization of the multi-mode PE. The inter-slice wires and registers for balancing cycles of units are not shown. (b) Activated data paths of the PE in various modes along with their functional descriptions. c in ❸ and ❹ is used for loading twiddle factors. C in ❷, ❺ and ❻ is used for gathering results. A in ❺ and ❻ is used for chaining multiple PE columns for pipelining EWE kernels.

network to minimize stalls caused by PE data loading and storing. For implementing the $2n_{row}$-point butterfly network required by (I)NTT, diagonal links are added to the first $\log_2 2n_{row}$ PE columns. Notably, the NTT decomposition involves twisting steps, i.e. element-wise modular multiplications, between executions of smaller (I)NTT kernels [11]. Therefore, additional columns of tailored PEs (TPEs), with the logic for butterfly operations removed, are incorporated. Each TPE performs one modular multiplication per cycle, necessitating two columns to match the throughput of the butterfly network. Previous approaches employ bidirectional butterfly networks to support both NTT and INTT, resulting in considerable area overheads [11]. Instead, our PEA consistently uses bit-reversed input and output orders, which are achieved through hardwired order reversing in the last two columns. This adjustment enables a unidirectional design, effectively reducing the inter-PE links needed by the butterfly network by half. (I)NTT can also reuse the data scatter&gather network to load twiddle factors.

We further design a *multi-mode PE* that supports both tensor and polynomial ring algebra. Tensor operations typically use low-bitwidth integers, whereas polynomial ring operations require at least 28-bit modular arithmetic [7]. In this paper, we use 8-bit precision for NPI due to its widespread adoption [14], and 32-bit precision for PI, aligning with the NPI bitwidth and following precedents in [25]. Fig. 5(a) illustrates that each PE is organized into 4×12 *slices*, each containing an 8-bit integer multiplier and two 16-bit integer adders (Ⅰ). The adders can also be configured as subtractors. *By dynamically orchestrating these slices through configurable data paths, the proposed PE seamlessly switches between two forms: tensor PE and ring PE (**I2**).* As shown in Fig. 5(b), when functioning as a tensor PE for GEMM, its slices operate independently in *MAC* mode (❶). Each slice cascades two 16-bit adders to construct a 32-bit adder, performing 8-bit multiplication and accumulation with operand forwarding. Consequently, the PE acts as a 4×12 output-stationary sub-systolic array, cooperating with the global systolic array dataflow of PEA. Note that other computation dataflows, such as input and weight stationary, can be realized by changing connections between inputs and computation units [29]. As depicted in Fig. 5(a), to construct a ring PE, we aggregate low-bitwidth multipliers and adders distributed in slices step by step. Initially, 32-bit multipliers are created by combining sixteen 8-bit multipliers and twenty-four 16-bit adders across 4×4 slices using the schoolbook method (Ⅱ). For our PE design, sophisticated methods such as Karatsuba multiplication are undesirable due to their marginal resource savings [30]. By cascading the remaining 16-bit adders in slices, we also obtain 32-bit adders/subtractors. With these 32-bit integer units, we further form 32-bit modular arithmetic units including the Barrett reduction-based multiplier[1] [31] (Ⅲ), $1/2$ multiplier (Ⅳ), adder (Ⅴ), and subtractor (omitted here). Ultimately, the versatile ring PE (Ⅵ) is realized with these components. As shown in Fig. 5(b), by setting up corresponding data paths through multiplexers, it can switch between five PI kernel modes: ❷ *MMAC* for BConv. Each PE performs 32-bit modular multiplication and accumulation, with the entire PEA using the systolic array dataflow for modular multiplication between base table and polynomial matrices. ❸/❹ *CTB*/*GSB* for NTT/INTT. Each PE performs a Cooley-Tukey (CT)/Gentleman-Sande (GS) butterfly operation. We adopt a compact design to avoid overheads of pre-processing and post-processing in negative wrapped convolution-based NTT [28]. The entire PEA is organized as the butterfly network dataflow. ❺/❻ *MM*/*MA* for EWE-MM/EWE-MA. Each PE acts as a 32-bit modular multiplier/adder, and the PEA employs the SIMD dataflow. To minimize area overheads, TPEs remove unnecessary circuits required solely for *CTB* and *GSB* modes. Notably, our method is general and can easily adapt to other bitwidth settings.

Reusing PEAs across NPI and PI raises concerns regarding resource utilization. For GEMM, the PEA's equivalent systolic array size is $4n_{row} \times 12n_{col}$. For (I)NTT, the butterfly network structure requires $n_{col} - 2 = \log_2 2n_{row}$ [24]. By judiciously sweeping candidate parameters (see §IV-B4), we set $n_{row} = 32$ and $n_{col} = 8$ to balance the efficiencies of NPI and PI. The size of the resulting RTC is significantly smaller than the computation units used in prior FHE accelerators, which have 256 lanes [7], [11]. This necessitates a multi-core design (see §III-B2) for performance scaling. Additionally, it renders the canonical 2D decomposition method [25] unusable for the common $N = 2^{16}$, leading to the adoption of 3D NTT [23].

DLT: The proposed DLT executes *all essential data layout transformation kernels* delineated in §II-B, namely Perm, Trans, and Auto. Initially, Perm is converted into a series of 8-bit Trans, as outlined in [32]. We then integrate 8-bit Trans, 32-bit Trans, and 32-bit Auto into a vectorized swapping network that utilizes $2n_{row}$ lanes to meet the throughput requirements of (I)NTT computations on the PEA. *Our design unifies these kernels through two innovations: i) a modified swapping order and ii) low-bitwidth subnets (**I1**, **I2**).* To align with the network topology, the DLT shown in Fig. 4(b) employs a bottom-up quadrant swapping strategy for Trans using multiplexers and buffers, which is contrary to the top-down method

[1]It actually introduces integer units with non-uniform data widths, such as 32 bits and 33 bits [10]. This can be addressed by using 9-bit multipliers and 17-bit adders in specific slices, omitted here for simplicity.

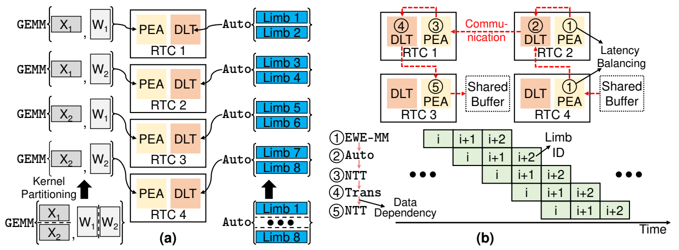

Fig. 6. (a) Intra-kernel parallelization of GEMM and Auto. (b) Resource allocation and pipeline diagram for inter-kernel parallelization of PI kernels.

used in [25]. For example, transposing an 8×8 matrix proceeds in three pipelined stages: quadrant swapping first on sixteen 2×2 sub-matrices (Stage 1), then on four 4×4 sub-matrices (Stage 2), and finally on the entire 8×8 matrix (Stage 3). Auto execution follows [11], where quadrant swapping is guided by precomputed control signals derived from rotation amounts. Additionally, to accommodate varying data widths, the network can be partitioned into four independent 8-bit subnets for 8-bit Trans in NPI, and combined into a 32-bit monolithic network for 32-bit Trans and Auto in PI.

2) Multi-Core Parallelization: RTPU integrates numerous RTCs to deliver performance on par with dedicated FHE accelerators, unlocking extensive parallelization opportunities. As shown in Fig. 6(a), it achieves intra-kernel parallelization by partitioning a kernel across multiple RTCs and executing all portions concurrently. For NPI kernels, input tensors are evenly split along their axes. To minimize communication between partitioned PI kernels, we use coefficient-based partitioning for BConv and limb-based partitioning for others [33]. However, the RTPU architecture faces limitations with solely intra-kernel parallelization. The primary drawback is the underutilization resulting from the mismatch between the limited parallelism of individual kernels and the abundance of small RTCs. For example, executing Auto with 8 limbs results in 50% idle cores when 16 RTCs are assigned. Additionally, the sequential execution of kernels with data dependencies necessitates a communication process: producer RTC→shared buffer→consumer RTC. This imposes high bandwidth and capacity pressure on the shared buffer. To address these issues, RTPU supports inter-kernel parallelization by mapping multiple kernels to RTCs simultaneously and creating a fine-grained pipeline. Specifically, NPI kernels are pipelined at microbatch granularity [26]. Most PI kernels use limb-based granularity. However, BConv requires all limbs rather than just one, preventing it from leveraging inter-kernel parallelism and forcing it to wait until the producer kernel completes. As illustrated in Fig. 6(b), the five-stage pipeline reads and writes the shared buffer at the beginning and end, significantly reducing intermediate buffer accesses. *Notably, intra-kernel parallelization remains necessary to balance the latencies of stages for optimizing pipeline efficiency*. For instance, ① EWE-MM is allocated two PEAs, each acting as an n_{row}-lane SIMD, to match the $2n_{row}$ throughput of other kernels. The RTPU scheduler incorporates both intra-kernel and inter-kernel parallelization to reduce kernel latencies and enhance resource utilization.

*3) Memory and NoC Constructions: The on-chip memory is efficiently shared between NPI and PI by exploiting their similar data reuse patterns and a configurable multi-bank structure (**I3**).* The global shared buffer is capable of prefetching input data that exceeds the capacities of local buffers, which include layer weights for NPI and evaluation keys for PI. Within each RTC, the local buffer enhances data reuse by storing tiled weights and feature maps for NPI, as well as tiled limbs for PI. We further organize the local buffer into 8-bit banks. When executing PI kernels, it combines every four banks into a 32-bit bank to match the bitwidth of PEA and DLT. Moreover, it can be bypassed to directly chain multiple RTCs, as depicted in Fig. 4, eliminating redundant local buffer accesses.

As depicted in Fig. 3(b), our design employs a fat tree-based intra-cluster NoC that efficiently supports two essential communication patterns for NPI and PI workloads: i) unicast and multicast traffic generated by tensor and polynomial partitioning in intra-kernel parallelization [11], [26], and ii) peer-to-peer communication necessary for pipeline construction in inter-kernel parallelization. Moreover, compared with the commonly used mesh topology offering placement-sensitive inter-RTC bandwidth, the inherent symmetry of fat tree topology provides a location-agnostic bandwidth guarantee [34]. This architectural property eliminates the need for complex placement optimization [35] and enables straightforward logical RTC mapping during online scheduling (see §III-C2).

C. Two-Stage RTPU Scheduler

RTPU scheduler dynamically allocates resources including the RTC regions, off-chip bandwidth, and shared buffer sizes for both NPI and PI tasks. Existing methods [12], [13] rely on tensor-specific performance models that are unable to characterize the polynomial ring-based computations in PI. Moreover, they are limited to intra-kernel parallelization and thus fail to leverage the extensive parallelization opportunities offered by massive RTCs. To overcome these shortcomings, we unify plaintext-based and ciphertext-based NN models into directed acyclic graphs (DAGs), where nodes and edges represent kernels and data dependencies, respectively. To enable inter-kernel parallelization, each DAG is sequentially executed at the granularity of a *kernel group* comprising up to n_{grp} kernels that create the pipeline. We then develop a unified performance model and two-stage scheduling strategy as detailed below.

1) Unified Performance Model: By overlapping computation and memory access, we model the execution time of the i-th task as the maximum of computation time (ct_i) and memory access time (mt_i), achieving a balance between estimation accuracy and implementation complexity. For its j-th kernel group, due to the pipelined execution of kernels, the computation time ct_j^i is calculated with:

$$ct_j^i = \underbrace{\sum_{k=1}^{n_{grp}} ct_{j,k}^i}_{\text{Latency of initialization}} + \underbrace{(n_{iter}-1)\cdot \max(\{ct_{j,k}^i\}_{1\le k\le n_{grp}})}_{\text{Latency of subsequent iterations}} \quad (3)$$

where $ct_{j,k}^i$ denotes the k-th kernel's computation time. For NPI kernels, n_{iter} represents the microbatch count, whereas for PI kernels (excluding BConv), it refers to the number of limbs. The computation time of a vectorized PI kernel depends on three factors: input polynomial dimension, hardware unit configuration, and degree of intra-kernel parallelization. For example, an Auto kernel with l-limb inputs requires $Nl/2n_{row}/n_{DLT}$ cycles, where these terms represent the polynomial coefficient count, DLT lane quantity, and number of allocated DLTs, respectively. Both GEMM and BConv utilize the systolic array dataflow, with their latencies modeled according to [12]. Using Eq. (4), memory access time estimation focuses on off-chip data transfers, owing to the substantial disparity between off-chip and on-chip bandwidth. For each kernel group, its external data comprises two components: intermediate results exchanged with the shared buffer at the pipeline's initial and final stages, and input operands such as layer weights for NPI and evaluation keys for PI. The equation indicates that if the volume of external data ($data_j^i$) exceeds the capacity of the task's designated shared buffer ($sbuf_i$), off-chip transmission occurs under the assigned bandwidth (bw_i).

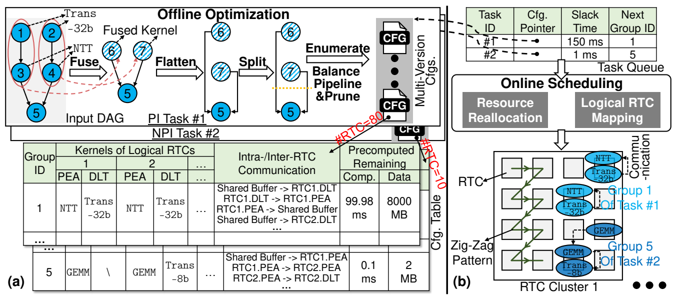

Fig. 7. Stages of (a) offline optimization and (b) online scheduling. Here `Trans-32b/8b` represents matrix transposition for 32/8-bit precision.

$$mt_j^i = \max\left(data_j^i - sbuf_i, 0\right)/bw_i \tag{4}$$

2) Scheduling Strategy: To minimize the hardware complexity of the online scheduler, we pregenerate multiple configuration *versions* offline and switch to the appropriate one at runtime [13].

Offline Optimization: As illustrated in Fig. 7(a), the input DAG initially undergoes rule-based kernel fusion to enhance hardware utilization. Certain kernel combinations, such as two consecutive `EWE` kernels, are fused for intra-PEA pipelining. Also, kernels utilizing PEAs (e.g. `NTT`) and DLTs (e.g. `Trans`) are fused into the same RTC. The DAG is then flattened into a sequence through depth-first topological sorting to minimize the memory footprint of intermediate results. This is crucial for PI which generates large intermediate polynomials [7]. The sequence is further split into multiple kernel groups of fixed size n_{grp}, with an exception for `BConv`, which lacks inter-kernel parallelism and is grouped individually to prevent pipeline stalls. Although resource-aware splitting [36], [37] can improve performance, it is unsuitable for the multi-tenant RTPU due to its inconsistent kernel groups across configuration versions, requiring complex rollback mechanisms for version switching. The final stage enumerates all possible RTC quantities and generates corresponding configuration versions. For each group, the degree of intra-kernel parallelization, i.e. the number of RTCs assigned to each fused kernel, is adjusted to balance pipeline stages. Consequently, each partitioned kernel is assigned to a *logical RTC*. To avoid resource over-provisioning, versions with marginal computation time improvements are pruned, enabling power-gating of unused RTCs. For example, tasks #1 and #2 in Fig. 7(a) use up to 80 and 10 RTCs, respectively.

Fig. 7(a) illustrates the detailed structure of configuration tables. Each entry corresponds to a kernel group, specifying the fused and partitioned kernels contained in each logical RTC. It also outlines the communication between DLTs and PEAs within the same logical RTCs or across multiple ones. To minimize the overhead of runtime performance estimation, each entry includes the precomputed remaining computation time and data volumes of subsequent kernel groups, calculated by summing ct_j^i in Eq. (3) and $data_j^i$ in Eq. (4). Although the multi-version approach results in bloated configuration files, we address this issue by storing them in off-chip memory and loading only the active table entries onto the chip.

Online Scheduling: As shown in Fig. 7(b), the task queue maintains each task's slack time before the QoS deadline, a pointer to multi-version configurations, and the next kernel group ID. *Notably, the online scheduling does not assume specific task arrival patterns and relies solely on the task queue status*. Details are provided below.

Resource reallocation occurs when new inference tasks arrive or existing tasks complete, adjusting three critical resources: RTCs, off-chip memory bandwidth, and shared buffer. The scheduler first identifies the minimum RTC requirement for each task by sequentially checking configuration versions until finding one with a remaining computation time less than the slack time. If RTCs are insufficient, tasks are allocated their minimum required RTCs in the order of urgency scores, defined as the ratio of remaining computation time to slack time. Tasks failing to receive RTCs are suspended. If there are sufficient RTCs, surplus cores are distributed to all tasks in proportion to their urgency scores. After quantitative allocation, RTC regions are decided using a zig-zag pattern. This simple heuristic prioritizes high-bandwidth local cluster and minimizes intra-cluster communication hops by positioning same-task RTCs under their nearest common ancestor routers in the fat tree topology. Off-chip memory bandwidth is then allocated proportionally to the peak demand of each task, derived from the ratio of remaining data volume to slack time. Next, the scheduler obtains the minimum shared buffer requirements based on precomputed remaining data volume and accumulated Eq. (4), where the max operation is relaxed to simplify solving. Shared buffer sizes are decided in a manner similar to the RTC allocation. At the end of resource reallocation, running tasks release their occupied RTCs and shared buffers once current kernel groups complete. For tasks with modified RTC quantities, new configuration table entries also need to be loaded. Active tasks then execute their subsequent kernel groups under the adjusted resource allocations.

Logical RTC mapping is triggered each time a kernel group entry in the configuration table executes. As shown in Fig. 7(b), logical RTCs are initially mapped to the allocated physical region, still following the zig-zag pattern. The relevant registers of PEAs and DLTs are then configured to execute their designated kernels. To further establish inter-RTC communication, NoC interfaces in the RTCs are set up to align the indices of logical and physical RTCs. Finally, the slack time and next group ID in the task queue are updated.

TABLE I
DESCRIPTION OF RTPU PARAMETERS.

Core Configuration	
PEA/DLT Dimension	$n_{row} = 32$, $n_{col} = 8$
Local Buffer Size	1 MB
System Configuration	
# RTC	4 clusters, 16 RTCs each
Shared Buffer Size	128 MB
NoC	Crossbar/Fat tree for inter-/intra-cluster NoC, 256-byte links
Off-Chip Memory Bandwidth	2 HBM2 [38], 500 GB/s each
Operating Frequency	1 GHz
Scheduler Setting	$n_{grp} = 6$

IV. Evaluation

A. Experimental Setup

Hardware Implementation. We set parameters of RTPU as outlined in Table I to align with the performance of contemporary FHE accelerators [7]. We implement the RTC using the ASAP7 7nm predictive process design kit [39], with synthesis and place-and-route performed by Cadence Genus and Cadence Innovus. The NoC overheads are modeled using DSENT [40]. The peak power of RTPU is estimated based on usage cases that activate all components.

Benchmark Workloads. We select five classic NN models from prior studies [7], [41]: MLP, LeNet-5, ResNet-20, SqueezeNet and MobileNet. Input image sizes are configured as 224×224 for NPI and 32×32 for PI. The NN compiler [42] converts these models into NPI DAGs, while the FHE compiler [41] produces PI DAGs at 128-bit security level (polynomial degree $N = 2^{16}$). We further construct two types of workloads: *single* workloads running tasks exclusively on the entire accelerator, and *mixed* workloads with randomized task

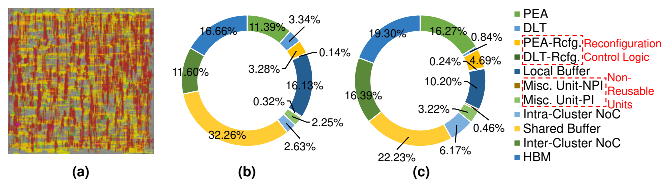

Fig. 8. (a) RTC layout. Breakdowns of (b) area and (c) peak power for RTPU.

arrivals to simulate concurrent servicing. For mixed workloads, the total number of tasks is fixed at 10^6, and we use varying PI-to-NPI task ratios to model diverse user distributions. QoS constraints for NPI tasks are set directly according to [43], whereas those for PI tasks are scaled based on their slowdown.

Baseline Architectures. We choose TPU [6] and SHARP [7] as baseline accelerators for NPI and PI, respectively. Given the significant resource disparities between the original TPU and RTPU, we maintain identical core quantities and I/O interface configurations. To model the dedicated accelerator-based deployment, we create a heterogeneous accelerator platform (HAP) integrating a TPU chip and a SHARP chip [7]. To ensure a fair comparison, we adjust the number of cores in TPU and the number of clusters in SHARP, reducing the area of both chips to nearly half that of RTPU. In addition, to compare against general-purpose hardware, we employ the NVIDIA A100 GPU with the CKKS implementation from [9].

Baseline Schedulers. We use the time-sharing PREMA [12] and spatial-sharing Planaria [13]. Their performance models are extended with the methods illustrated in §III-C1 to support PI kernels.

Simulation Infrastructure. We develop a cycle-accurate simulator in Python, which encompasses latency and throughput models of computation units, buffers and NoCs of RTPU.

Evaluation Metrics. To assess reconfiguration efficiency, we report: i) the area and power overheads of control logic, and ii) the area proportion of hardware units shared between NPI and PI. We consider global average latencies of tasks, measuring each task's latency from arrival to completion. Following [9], we quantify GPU utilization with SM occupancy. Notably, this single metric may not directly correlate with the achieved throughput [44]. For other architectures, utilization is assessed through the area proportion of active computation units. Energy efficiency is quantified using the energy-delay product (EDP).

B. Experimental Results

1) Implementation Results: Fig. 8(a) presents the RTC layout. The entire RTPU achieves 177.63 mm^2 area and 164.74 W peak power. The area and power breakdowns shown in Fig. 8 (b) and (c) highlight two key points: i) ***The control logic for RTPU reconfiguration incurs low overheads***. PEA-Rcfg. and DLT-Rcfg., involving additional multiplexers and registers within PEAs and DLTs, constitute merely 3.42% of total area and 4.93% of power consumption. This is minor compared with the significant flexibility they offer. ii) ***The RTPU achieves high silicon reuse between NPI and PI***. General units like buffers and NoCs dominate the overall area. For a PE, all 48 multipliers and 96 adders are used in the tensor PE configuration, while 48 multipliers and 94 adders are repurposed for the ring PE configuration. Thus, nearly the entire PEA can be reused between NPI and PI. Moreover, the DLT is completely shared by NPI and PI kernels. Components used solely by NPI/PI, i.e. NPI/PI-dedicated miscellaneous units, only account for 2.57% of the area, allowing 97.43% of hardware resources to be reused between NPI and PI.

2) Architecture Comparisons: First, we compare different architectures under single workloads to understand their performance gaps, as shown in Fig. 9. On average (geometric mean), RTPU exhibits

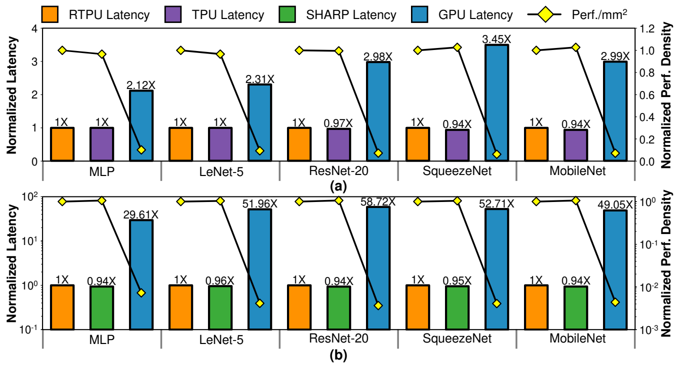

Fig. 9. Comparisons of latency and performance density between RTPU and baseline architectures using single workloads of (a) NPI and (b) PI. All results are normalized to those of RTPU.

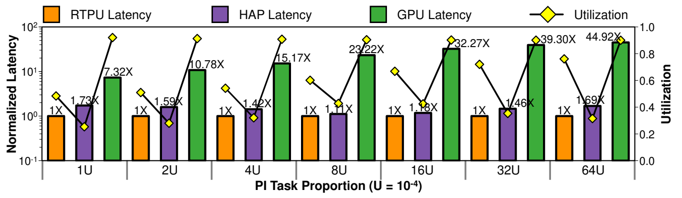

Fig. 10. Comparisons of latency and computation unit utilization between RTPU and baseline architectures using mixed workloads across varying PI task proportions. Latency results are normalized to those of RTPU.

only 1.03× higher latency than the NN-specific TPU due to their similar systolic array dimensions (128×96 versus 128×128). For smaller models like MLP and LeNet-5, the larger systolic arrays in TPU are underutilized, resulting in lower performance density. For PI tasks, RTPU achieves an average of 1.06× higher latency and 0.95× lower performance density compared to the FHE-specific SHARP. This gap arises from two factors: i) SHARP's 256-lane ($= \sqrt[2]{N}$ for 2D NTT) configuration fully utilizes its NTT units, while RTPU's 64-lane ($\neq \sqrt[3]{N}$ for 3D NTT) design leaves 8.33% of the PEA and DLT lanes idle during `(I)NTT` execution. ii) SHARP uses heterogeneous computation units like NTT and BConv units, which lead to inferior global utilization with imbalanced kernel distributions. In contrast, RTPU's homogeneous RTC design allows flexible core allocation, mitigating the disadvantages noted in factor i). RTPU outperforms the GPU across all models, with average latency/performance density improvements of 2.73×/12.71× for NPI tasks and 47.17×/219× for PI tasks. While the GPU incorporates Tensor Cores to optimize tensor computation, it lacks native support for modular arithmetic and NTT dataflow [9], [45]. This limitation necessitates algorithmic modifications, making it less efficient in handling polynomial rings.

Second, we evaluate architectural flexibility using mixed workloads simulating real-world deployments. As shown in Fig. 10, RTPU delivers an average speedup of 1.44× over HAP. Despite incorporating optimized accelerators (TPU and SHARP), HAP suffers from resource idling when handling imbalanced NPI and PI tasks due to the architectural divergence between these components. Consequently, its performance initially improves but then degrades as the proportion of PI tasks increases, a trend following the utilization curve. Given the three-order-magnitude latency gap between performing NPI on TPU and PI on SHARP, HAP reaches peak performance when PI tasks occupy 0.08% of the total tasks. *Regardless of the TPU/SHARP area ratio, HAP consistently exhibits low utilization with mismatched NPI and PI tasks, fundamentally limiting its adaptability to uncertain user distributions*. Furthermore, RTPU achieves an average of 20.62× speedup over GPU. While GPU maintains high utilization

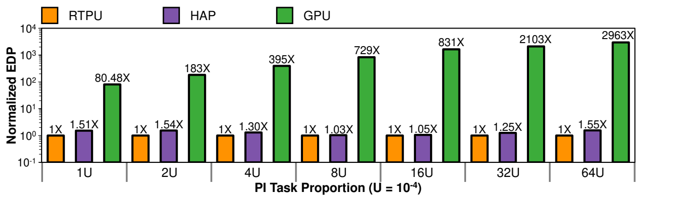

Fig. 11. Comparison of EDP between RTPU and baseline architectures using mixed workloads across varying PI task proportions. Results are normalized to those of RTPU.

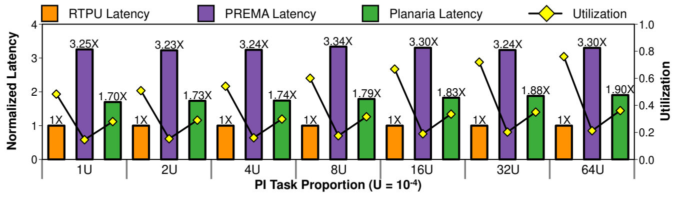

Fig. 12. Comparisons of latency and computation unit utilization between RTPU and baseline schedulers using mixed workloads across varying PI task proportions. Latency results are normalized to those of RTPU.

(over 90%), it suffers from inherent computational inefficiencies, particularly for PI-dominated workloads. Notably, RTPU utilization increases with the PI task proportion. This occurs because PI kernels (with limb-based partitioning) leverage RTCs more efficiently than NPI kernels (with tensor-based partitioning) under our hardware parameters and NN model settings. To illustrate, when executing median-size kernels (e.g. NTT with 20-limb and GEMM with $(288 \times 11664, 11664 \times 92)$-tensor inputs) and increasing the allocated RTC quantity from 1 to 10, PI maintains an average PEA utilization of 0.83, significantly higher than the 0.43 utilization achieved by NPI.

Third, we compare energy efficiency of different architectures using mixed workloads, with results illustrated in Fig. 11. Although RTPU consumes slightly more energy than HAP due to its reconfiguration logic, its significant latency reduction results in an average of 1.30× lower EDP. Additionally, RTPU substantially outperforms GPU in both energy and latency, improving EDP by 580× on average.

Based on the above evaluations, we conclude that ***the RTPU architecture demonstrates comprehensive advantages in performance, flexibility, and energy efficiency over both dedicated accelerators and general-purpose hardware***.

3) Scheduler Comparisons: We compare different scheduling strategies using mixed workloads, with results depicted in Fig. 12. RTPU scheduler achieves average speedups of 3.27× and 1.79× over PREMA and Planaria, respectively. PREMA's kernel-wise strategy partitions each kernel across all RTCs for sequential execution. Its maximized intra-kernel parallelization leads to small operand sizes, resulting in severely underutilized PEAs and DLTs (utilization below 21.24%). Planaria mitigates this by running multiple tasks concurrently, reducing intra-kernel parallelization by assigning fewer RTCs per task. Nevertheless, this approach is still fundamentally constrained to kernel-wise scheduling, which prevents it from fully exploiting the abundant RTCs in RTPU, especially under insufficient concurrent tasks. Our RTPU scheduler operates at the granularity of kernel groups, strategically reducing intra-kernel parallelism requirements through inter-kernel pipelining. Therefore, we conclude that ***prior schedulers are ill-suited to the RTPU architecture, whereas our approach enables significantly improved resource utilization***.

4) Design Space Exploration: We first analyze the RTC size parameter n_{row} under fixed area budgets for the RTCs and NoC. Fig. 13(a) shows that $n_{row} = 32$ yields optimal global average latency for RTPU, rendering the canonical setting of $n_{row} = 128$

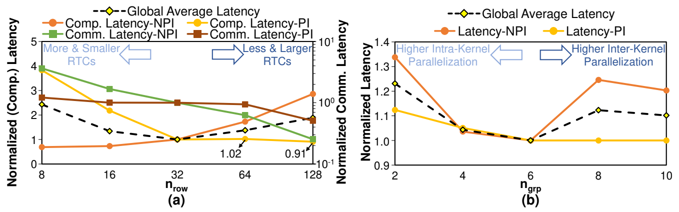

Fig. 13. Comparisons of different (a) n_{row} and (b) n_{grp}, where their results are normalized to the counterparts of $n_{row} = 32$ and $n_{grp} = 6$, respectively. Both (a) and (b) are conducted under mixed workloads containing 0.08% PI tasks, ensuring balanced execution time of NPI and PI kernels.

[7] suboptimal. Smaller n_{row} values (i.e. 8 and 16) create excessive RTCs, making the system communication-bound due to the limited scalability of the fat tree-based intra-cluster NoC [46]. Note that RTC bandwidth requirements differ when executing NPI versus PI kernels, with NPI tasks undergoing a more drastic change in communication latency. Conversely, larger n_{row} values (i.e. 64 and 128) result in fewer RTCs. They significantly alleviate NoC pressure but shift the system bottleneck to computation. NPI computation latency increases because GEMM achieves diminishing utilization on larger systolic arrays. PI tasks instead show fluctuating computation latency, which is caused by the decomposition of (I)NTT kernels (see §II-B2). The utilization of RTC varies depending on the mismatches between its lane number ($2n_{row}$) and decomposed kernel sizes ($N_I/N_J/...$). Overall, ***we set n_{row} to 32 to achieve a computation-communication trade-off for both NPI and PI kernels***.

We then study the scheduler parameter n_{grp} that controls the maximum kernel group size. As depicted in Fig. 13(b), both NPI and PI tasks initially experience performance improvements as n_{grp} increases. This is because running multiple kernels concurrently alleviates the underutilization of computation units caused by excessive intra-kernel parallelization. However, these benefits diminish when n_{grp} continues to grow. Note that for NPI, the sizes of layers within NN models vary significantly. Balancing pipeline stages becomes more difficult for larger kernel groups, since each layer requires at least one PEA/DLT regardless of its size. This constraint introduces substantial pipeline bubbles and latency penalties when n_{grp} exceeds 6. For PI tasks, the actual kernel group sizes also depend on BConv (see §III-C2), which frequently appears and mitigates the effects of further increasing n_{grp}. Therefore, ***we set n_{grp} to 6 to strike a balance between intra-kernel and inter-kernel parallelization***.

V. Discussion and Conclusion

A. Applicable Scenarios

After considering the benefits and overheads, we believe that RTPU is ideal for scenarios meeting three conditions: i) privacy-sensitive users mandating FHE for extreme data security, ii) equally important NPI and PI demands with temporal variations, and iii) tolerance for small performance and power overheads. In rare situations where these conditions are not met, the programmable architecture with high generality or dedicated accelerators optimized for performance and power efficiency might be necessary.

B. Conclusion

In this paper, we rethink the accelerator design paradigm for coexisting NPI and PI in data centers. We introduce the reconfigurable RTPU that bridges the gap between tensor-based plaintext and polynomial ring-based ciphertext computations by leveraging their inherent similarities. The proposed architecture delivers performance on par with dedicated accelerators while offering higher flexibility and incurring minor overheads. Furthermore, the scheduler efficiently allocates resources to tasks with varying privacy requirements.

References

[1] Q. Weng *et al.*, "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, 2022, pp. 945–960.

[2] J. Mo, *et al.*, "Towards fast and scalable private inference," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*, 2023, pp. 322–328.

[3] M. Creeger, "The rise of fully homomorphic encryption: Often called the holy grail of cryptography, commercial fhe is near." *Queue*, vol. 20, no. 4, pp. 39–60, 2022.

[4] L. d'Aliberti *et al.*, "Enable fully homomorphic encryption with amazon sagemaker endpoints for secure, real-time inferencing," 2023. [Online]. Available: https://aws.amazon.com/cn/blogs/machine-learning/enable-fully-homomorphic-encryption-with-amazon-sagemaker-endpoints-for-secure-real-time-inferencing/

[5] O. Soceanu *et al.*, "The ultimate tool for data privacy: Fully homomorphic encryption," 2022. [Online]. Available: https://research.ibm.com/blog/fhe-cloud-security-hE4cloud

[6] N. P. Jouppi *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. IEEE, 2021, pp. 1–14.

[7] J. Kim *et al.*, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[8] J. Hruska, "As chip design costs skyrocket, 3nm process node is in jeopardy," 2018. [Online]. Available: https://www.extremetech.com/computing/272096-3nm-process-node

[9] S. Fan *et al.*, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2023, pp. 922–934.

[10] Y. Yang *et al.*, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2023, pp. 870–881.

[11] J. Kim *et al.*, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2022, pp. 1237–1254.

[12] Y. Choi *et al.*, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2020, pp. 220–233.

[13] S. Ghodrati *et al.*, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2020, pp. 681–697.

[14] S. Alam *et al.*, "Survey of deep learning accelerators for edge and emerging computing," *Electronics*, vol. 13, no. 15, p. 2988, 2024.

[15] B. Reagen *et al.*, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2021, pp. 26–39.

[16] R. Mareta *et al.*, "A bootstrapping-capable configurable ntt architecture for fully homomorphic encryption," *IEEE Access*, 2024.

[17] J. Mu *et al.*, "Alchemist: A unified accelerator architecture for cross-scheme fully homomorphic encryption," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[18] S. Kim *et al.*, "Cifher: A chiplet-based fhe accelerator with a resizable structure," *arXiv preprint arXiv:2308.04890*, 2023.

[19] Y. Du *et al.*, "Chiplever: Towards effortless extension of chiplet-based system for fhe," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[20] X. Li *et al.*, "Performance analysis of gpu-based convolutional neural networks," in *2016 45th International conference on parallel processing (ICPP)*. IEEE, 2016, pp. 67–76.

[21] T. Norrie *et al.*, "Google's training chips revealed: Tpuv2 and tpuv3." in *Hot Chips Symposium*, 2020, pp. 1–70.

[22] J. H. Cheon *et al.*, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[23] S. Kim *et al.*, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

[24] J. Mu *et al.*, "Scalable and conflict-free ntt hardware accelerator design: Methodology, proof, and implementation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 5, pp. 1504–1517, 2022.

[25] N. Samardzic *et al.*, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[26] L. Zheng *et al.*, "Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation*, 2022, pp. 559–578.

[27] N. Neda *et al.*, "Multi-precision deep neural network acceleration on fpgas," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 454–459.

[28] N. Zhang *et al.*, "Highly efficient architecture of newhope-nist on fpga using low-complexity ntt/intt," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 49–72, 2020.

[29] M. Han *et al.*, "Redas: A lightweight architecture for supporting fine-grained reshaping and multiple dataflows on systolic array," *IEEE Transactions on Computers*, 2024.

[30] C. Rafferty *et al.*, "Evaluation of large integer multiplication methods on hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1369–1382, 2017.

[31] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[32] J. L. Jodra *et al.*, "Efficient 3d transpositions in graphics processing units," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 876–891, 2015.

[33] R. Agrawal *et al.*, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 685–697.

[34] W. J. Dally *et al.*, *Principles and practices of interconnection networks*. Elsevier, 2004.

[35] S. Murali *et al.*, "Bandwidth-constrained mapping of cores onto noc architectures," in *Proceedings design, automation and test in Europe conference and exhibition*, vol. 2. IEEE, 2004, pp. 896–901.

[36] S. Zheng *et al.*, "Efficient scheduling of irregular network structures on cnn accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3408–3419, 2020.

[37] X. Cai *et al.*, "Optimus: An operator fusion framework for deep neural networks," *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 1, pp. 1–26, 2022.

[38] JEDEC, "*High Bandwidth Memory DRAM*," Technical Report JESD235D, 2021.

[39] V. Vashishtha *et al.*, "Asap7 predictive design kit development and cell design technology co-optimization," in *2017 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2017, pp. 992–998.

[40] C. Sun *et al.*, "Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 2012, pp. 201–210.

[41] S. Cheon *et al.*, "{DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6993–7010.

[42] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[43] V. J. Reddi *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 2020, pp. 446–459.

[44] Elvinger *et al.*, "Measuring gpu utilization one level deeper," *arXiv preprint arXiv:2501.16909*, 2025.

[45] K. Shivdikar *et al.*, "Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 670–684.

[46] D. Ludovici *et al.*, "Assessing fat-tree topologies for regular network-on-chip design under nanoscale technology constraints," in *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2009, pp. 562–565.