REED: Chiplet-based Accelerator for Fully Homomorphic Encryption

Aikata Aikata[1], Ahmet Can Mert[1], Sunmin Kwon[2], Maxim Deryabin[2] and Sujoy Sinha Roy[1]

[1] Graz University of Technology, Graz, Austria
{aikata,ahmet.mert,sujoy.sinharoy}@tugraz.at
[2] Samsung Advanced Institute of Technology, Samsung Electronics, Korea
{sunmin7.kwon,max.deriabin}@samsung.com

Abstract. Fully Homomorphic Encryption (FHE) enables privacy-preserving computation and has many applications. However, its practical implementation faces massive computation and memory overheads. To address this bottleneck, several Application-Specific Integrated Circuit (ASIC) FHE accelerators have been proposed. All these prior works put every component needed for FHE onto one chip (monolithic), hence offering high performance. However, they encounter common challenges associated with large-scale chip design, such as inflexibility, low yield, and high manufacturing costs. In this paper, we present the *first-of-its-kind* multi-chiplet-based FHE accelerator 'REED' for overcoming the limitations of prior monolithic designs. To utilize the advantages of multi-chiplet structures while matching the performance of larger monolithic systems, we propose and implement several novel strategies in the context of FHE. These include a scalable chiplet design approach, an effective framework for workload distribution, a custom inter-chiplet communication strategy, and advanced pipelined Number Theoretic Transform and automorphism design to enhance performance.

Our instruction-set and power simulations experiments with a prelayout netlist indicate that REED 2.5D microprocessor consumes 96.7mm^2 chip area, 49.4 W average power in 7nm technology. It could achieve a remarkable speedup of up to 2,991× compared to a CPU (24-core 2×Intel X5690) and offer 1.9× better performance, along with a 50% reduction in development costs when compared to state-of-the-art ASIC FHE accelerators. Furthermore, our work presents the *first* instance of benchmarking an encrypted deep neural network (DNN) training. Overall, the REED architecture design offers a highly effective solution for accelerating FHE, thereby significantly advancing the practicality and deployability of FHE in real-world applications.

Keywords: Homomorphic Encryption, Hardware Acceleration, Chiplets, CKKS

1 Introduction

Data breaches can put millions of private accounts at risk because data is often stored or processed without encryption, making it vulnerable to attacks [IBM20, Mor20, LSL20]. Fully Homomorphic Encryption (FHE) is a solution that allows secure, private computations, communications, and storage. It enables servers to compute on homomorphically encrypted data and return encrypted outputs. FHE has a wide range of applications, including cloud computing [MNLK23, KKL+23a], data processing [BZP+23], and machine learning [PMSW18]. The concept of FHE was introduced in 1978 by Rivest, Adleman, and Dertouzos [RAD78], and the first FHE scheme was constructed in 2009 by Gentry [Gen09]. Since then, many FHE schemes have emerged- BGV [BGV11], FV [FV12], CGGI [CGGI20],

and CKKS [CKKS17, CHK+18b, KKL+23b]. These schemes allow computations to be outsourced without the need to trust the service provider, providing a functional and dependable privacy layer.

Despite significant progress in the mathematical aspects of FHE, state-of-the-art FHE schemes typically introduce 10,000× to 100,000× slowdown [JLK+21] compared to plaintext calculations. This overhead can be attributed to plaintext expanding into large polynomials when encrypted using an FHE scheme. Subsequently, simple operations, like plaintext multiplication, translate into complex polynomial operations. FHE's massive computation and data overhead hinders its deployment in real-life applications. To bridge this performance gap, researchers have proposed acceleration techniques on various platforms, including GPU, FPGA, and ASIC [BDTV23, WHEW14, RCK+20, KLK+22, SFK+22, KKK+22, GBP+23, FSK+21, KKC+23, TRG+20, XZH21, NSA+22, FWX+23, MAK+23, RLPD20, WH13, RJV+15, JKA+21, BHM+20, SRTJ+19, RJV+18]. Software implementations offer flexibility but poor performance. Attempts have been made to provide GPU [JKA+21, BHM+20] and FPGA-based solutions [MAK+23, RLPD20, SRTJ+19]. However, the performance gap is still 2-3 orders compared to plain computation.

Currently, the fastest hardware acceleration results for FHE have been reported using ASIC modeling [KLK+22, SFK+22, KKK+22, GBP+23, FSK+21, KKC+23]. The works propose utilizing large chip architecture designs with all FHE building blocks onto a single chip to maximize performance, hence monolithic. While simulations of these architectures show that they can achieve high performance for FHE workloads, the limitations of the current manufacturing capabilities, such as inflexibility, low yield, and higher manufacturing costs [Gon21], impact their real-world deployment. For instance, the large architectures [KKK+22, KLK+22, KKC+23] with area-consumption of approximately 400mm^2, result in a manufacturing yield of only 67% [MWW+22], chip fabrication cost of over 25 million US$ [MUS], and long time-to-market (>3 years).

Additionally, several of these proposals overlook the crucial need for communication-computation parallelism as the off-chip to on-chip communication is slower than the chip's computation speed. Our analysis shows that this feature is important in an FHE accelerator for achieving good performance when running complex tasks like neural network training. Prior works also utilize higher on-chip bandwidth due to readily available on-chip memory (20TB/s [KLK+22], 36TB/s [KKC+23], and 84TB/s [SFK+22]). Replacing this on-chip memory with cheaper HBM3 (1.2TB/s bandwidth) would require 17 to 70 HBM3 modules to match the necessary bandwidth.

In summary, while the large and complex monolithic FHE architectures proposed in prior works show promise, they face practical challenges such as high manufacturing costs, yield rates, and extended time-to-market. Addressing these challenges opens the door to exploring new approaches like chiplet-based architecture design. Chiplet-based architecture design utilizes multiple smaller chiplets instead of one large monolithic chip to realize a large system. Chiplets are modular building blocks that are combined to create more complex integrated circuits, such as CPUs, GPUs, Systems-on-Chip (SoCs), or System-in-Package (SiPs).

The transition to chiplet integrated systems represents both the present and future of architectural designs [Gon21, ZSB21, Man22, YLK+18, GPG23, MWW+22]. In the DATE2024 keynote talk [RD24], the speaker remarks how chiplet-based designs help 'push the performance boundaries, with maximum efficiency, while managing costs associated with manufacturing and yield'. Chiplet-based architectures also feature the advantage of tiling beyond the reticle limit ($858mm^2$) [GPG23] as multiple chiplets can be integrated for better performance. Although chiplet-based architectures enjoy the aforementioned advantages, they also face a trade-off between performance and yield. Multiple smaller chiplets offer high yields and reduced manufacturing costs but, at the same time, experience performance overhead due to slower chiplet-to-chiplet communication. Taking the advantages and

challenges of chiplet-based systems into consideration, we are curious to investigate the following research questions:

How can we design and optimize a multi-chiplet accelerator for FHE that matches the performance of large monolithic FHE accelerators while overcoming the inherent challenges of monolithic designs?

To investigate the question mentioned above, we present REED, a multi-chiplet architecture for FHE acceleration. We propose a holistic design methodology covering all aspects of FHE acceleration, from low-level building blocks to high protocol levels, and reduce the area to $43.9mm^2$ for one REED-chiplet. This includes the first scalable design methodology for one chiplet and ensures full utilization of chiplets for varying amounts of available off-chip data bandwidths. After finalizing an efficient design of one chiplet, we move to a data and task distribution study for multiple chiplets in the context of CKKS [CKKS17] routines. Towards this, we contribute novel strategies that offer long-term computation and communication parallelism. Finally, we synthesize the proposed design methodology for ASIC and report application benchmarks.

Contributions

To the extent of our knowledge, this is the *first chiplet-based architecture for accelerating FHE*. Throughout this work, we have followed Occam's razor, seeking the simplest solutions for the best results. We unfold our major contributions as follows:

- **Chiplet-based FHE accelerator:** We present a novel and cost-effective chiplet-based FHE implementation approach, which is inherently scalable[1]. The chiplets are homogeneous (i.e., identical), which reduces testing and integration costs. REED with 2.5D packaging surpasses state-of-the-art work SHARP_{64}[KKC+23] with 1.9× better performance and 2× less development cost.
- **Workload division strategy:** The first step to realizing a multi-chiplet architecture is to develop an efficient disintegration strategy that helps us divide the workloads among multiple chiplets and reduces memory consumption. Hence, we propose an interleaved data and workload distribution technique for all FHE routines.
- **FHE-tailored efficient C2C communication:** Chiplet-based architectures suffer from slow C2C (chiplet-to-chiplet) communication. We address this by proposing the *first non-blocking ring-based inter-chiplet communication* strategy tailored to FHE. This mitigates data exchange overhead during the KeySwitch macro-routine, accelerating Bootstrapping (the most expensive FHE routine).
- **Scalable design:** To attain scalability by design, we propose a configuration-based design methodology such that the memory read/write and computational throughput are the same. Changing the configuration parameters allows the architecture to adapt to the desired area and throughput requirements. This also offers inherent communication-computation parallelism in the design of every chiplet.
- **Novel compute acceleration:** Furthermore, we present new design techniques for the micro-procedures of FHE- the number-theoretic transform (NTT) and automorphism (AUT). Our approach introduces Hybrid NTT, eliminating the need for expensive transpose operation and scratchpad memory. It is easily scalable for higher or lower polynomial degrees. Hence, other applications, such as zero-knowledge

[1] Proof-of-concept implementation for multi-chiplet cycle accurate model, NTT/INTT unit, and Automorphism unit is open source and available at https://github.com/aikata10/REED/.

proofs, can also benefit from this, where transposition is expensive due to high polynomial degrees. Additionally, we have prototyped these building blocks on FPGA- AlveoU250.

- **Application benchmark:** Finally, we choose parameters offering high precision and good performance. REED is the *first work to benchmark an encrypted deep neural network training*, showcasing practical and real-world impact. While CPU (24-core, 2×Intel Xeon CPU X5690 @ 3.47GHz) requires 29 days to finish it, REED 2.5D would take only 15.4 minutes, a realistic time for an NN training. We also use DNN training to run accuracy/precision experiments and validate our parameter choice.

Connection and comparison with chiplet designs for ML

While prior chiplet-based Machine Learning (ML) works address similar problems, our solutions are tailored to meet FHE requirements more effectively. For instance, [SCV+21] addresses MCM's long "tail-latency" issue using non-uniform work distribution and communication-aware data placement. In the context of FHE, we resolve this by running parallel computations over extended periods, ensuring uniform task distribution and data placement. Our chiplet interconnections are ring-like and unidirectional. Although we do not propose an automatic tool, our analysis, similar to [TCDM21], focuses on long-term chiplet utilization based on FHE's computational-depth. Our methodology introduces a new configuration-based design built from scratch with novel building blocks and high-level protocols. In contrast to [HKKR20], which combines heterogeneous-chiplets, we propose homogeneous-chiplets observing unique data-flow of FHE. A common limitation of the prior works is that they propose very small chiplet sizes (2 to 6 mm^2), which is too small as per a recent study done by the authors in [GPG23]. Thus, we ensure that our chiplet sizes fall within the optimal range.

2 Background

Let $\mathbb{Z}_Q$ represent the ring of integers in the $[0, Q-1]$ range. $\mathcal{R}_{Q,N} = \mathbb{Z}_Q[x]/(x^N+1)$ refers to polynomial ring containing polynomials of degree at most $N-1$ and coefficients in $\mathbb{Z}_Q$. In the Residue Number System (RNS) [Gar59] representation, Q is a composite modulus comprising co-prime moduli, $Q = \prod_{i=0}^{L-1} q_i$. The RNS representation is used to divide a big computation modulo Q into much smaller computations modulo q_i such that the small computations can be carried out in parallel. With the application of RNS, a polynomial $a \in \mathcal{R}_{Q,N}$ becomes a vector, say $\boldsymbol{a}$, of residue polynomials. Let the i-th residue polynomial within $\boldsymbol{a}$ be denoted as $a^i \in \mathcal{R}_{q_i,N}$. We use the 'monotype' font (`c`/`sk`) to represent ciphertexts/keys. Operators $\cdot$ and $\langle , \rangle$ denote the multiplication and dot-product between two ring elements. Noise (e) is refreshed for every computation. The tilde sign ($\tilde{}$) represents a data in NTT format (e.g., $\texttt{NTT}(a) = \tilde{a}$).

2.1 FHE schemes and CKKS routines

Different FHE schemes exist in literature, such as, BFV [FV12], BGV [BGV11], CGGI [CGGI20], and CKKS [CKKS17, CHK+18b]. These schemes use polynomial arithmetic but differ primarily in the data types they can encrypt. For instance, BGV and BFV encrypt integers, while CKKS encrypts fixed-point numbers. Due to the support for fixed-point arithmetic, CKKS is widely adopted for benchmarking machine learning applications [HHCP19, KSK+18]. Therefore, this work targets the RNS (Residue Number System) CKKS [CHK+18b]. Other FHE schemes like BGV and B/FV are also based on RLWE and require similar operations as in CKKS. Thus, these schemes can utilize the

Table 1: CKKS Parameters

Parameter	Definition
$N, n\ (\leq \frac{N}{2})$	Polynomial size, maximum slots packed
Q, q_i	Coefficient modulus, RNS bases $Q = \prod_{i=0}^{L} q_i$
L, l	Multiplicative depth (#RNS bases - 1) $l < L$
$dnum$	Number of digits in the switching key
P, p_i	Special modulus and its RNS base
$K\ (= \lceil \frac{L+1}{dnum} \rceil)$	Number of RNS bases for $P = \prod_{i=0}^{K-1} p_i$
w	Word size $(\log p_i, \log q_i)$
L_{boot}, L_{eff}	Multiplicative depth of/after bootstrapping

same design methodology for varying parameters. In the following, we briefly describe the main procedures within the RNS CKKS [CHK+18b, KPP22, HK19] for ciphertexts at level l (multiplicative depth is l) where $l < L$, $Q_l = \prod_{i=0}^{l} q_i$, and L is the maximum level. The residue polynomial associated with each modulus q_i in the RNS representation is commonly called the RNS limb. A CKKS ciphertext consists of components, e.g., $\mathtt{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1)$, where $\boldsymbol{c}_0$ and $\boldsymbol{c}_1$ are vectors of limbs. Table 1 describes the CKKS parameters, and algorithmic descriptions are provided for $dnum = L + 1 (K = 1)$.

1. $\mathtt{CKKS.Add}(\mathtt{c}, \mathtt{c}')$: As shown in Algorithm 1, this operation takes two input ciphertexts $\mathtt{c}$ and $\mathtt{c}'$ and computes $\mathtt{c}_{\text{add}} = (\boldsymbol{d}_0, \boldsymbol{d}_1) = (\boldsymbol{c}_0 + \boldsymbol{c}'_0, \boldsymbol{c}_1 + \boldsymbol{c}'_1)$.

2. $\mathtt{CKKS.Mult}(\mathtt{c}, \mathtt{c}')$: It multiplies the two input ciphertexts $\mathtt{c}$ and $\mathtt{c}'$, as shown in Algorithm 3, and computes the non-linear ciphertext $\mathtt{d} = (\boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2) = (\boldsymbol{c}_0 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_0 \cdot \boldsymbol{c}'_1 + \boldsymbol{c}_1 \cdot \boldsymbol{c}'_0, \boldsymbol{c}_1 \cdot \boldsymbol{c}'_1)$. Subsequently, $\mathtt{CKKS.KeySwitch}$ transforms $\mathtt{d}$ into a linear ciphertext. The computation is done on data in NTT format.

Algorithm 1 $\mathtt{CKKS.Add}$ [CHK+18b]

In: $\mathtt{c} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$, $\mathtt{c}' = (\tilde{\mathbf{c}}'_0, \tilde{\mathbf{c}}'_1) \in R^2_{Q_l}$
Out: $\mathtt{c}_{\text{add}} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \in R^2_{Q_l}$
1: $\tilde{\mathbf{d}}_0 \leftarrow \tilde{\mathbf{c}}_0 + \tilde{\mathbf{c}}'_0$, $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{c}}_1 + \tilde{\mathbf{c}}'_1$

Algorithm 2 $\mathtt{CKKS.Rotate}$ [CHK+18b]

In: $\mathtt{c} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in R^2_{Q_l}$, rot
Out: $\mathtt{c}_{\text{rot}} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \in R^2_{Q_l}$
1: $(\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1) \leftarrow \rho_{rot}(\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1)$

Algorithm 3 $\mathtt{CKKS.Mult}$ [CHK+18b]

In: $\mathtt{ct} = (\tilde{\mathbf{c}}_0, \tilde{\mathbf{c}}_1) \in R^2_{Q_l}$
In: $\mathtt{ct}' = (\tilde{\mathbf{c}}'_0, \tilde{\mathbf{c}}'_1) \in R^2_{Q_l}$
Out: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$

1: $\tilde{\mathbf{d}}_0 \leftarrow \tilde{\mathbf{c}}_0 \cdot \tilde{\mathbf{c}}'_0$
2: $\tilde{\mathbf{d}}_2 \leftarrow \tilde{\mathbf{c}}_1 \cdot \tilde{\mathbf{c}}'_1$
3: $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{c}}_0 \cdot \tilde{\mathbf{c}}'_1$
4: $\tilde{\mathbf{d}}_1 \leftarrow \tilde{\mathbf{d}}_1 + \tilde{\mathbf{c}}_1 \cdot \tilde{\mathbf{c}}'_0$

3. $\mathtt{CKKS.Rotate}(\mathtt{c}, rot, \mathtt{ksk}_{\mathtt{rot}})$: It rotates the plaintext slots within $\mathtt{c}$ by rot. First, a permutation ρ is applied to the ciphertext polynomial coefficients, as shown in Algorithm 2. This permutation is called automorphism and is determined by the Galois element $gle = 5^{rot} \bmod 2N$. Finally, the permuted ciphertext is processed by $\mathtt{CKKS.KeySwitch}$ using the rotation key $\mathtt{ksk}_{\mathtt{rot}}$.

4. $\mathtt{CKKS.KeySwitch}(\mathtt{d}, \mathtt{ksk})$: It uses a KeySwitch or evaluation key $\mathtt{ksk}$ to homomorphically transform a ciphertext decryptable under one key into a new ciphertext decryptable under another (original) key, as illustrated in Algorithm 4. It computes $\mathtt{c}''$ where $\boldsymbol{c}''_0 = \sum_{i=0}^{l-1} d_2^i \cdot ksk_0^i \in \mathcal{R}_{PQ_l,N}$ and $\boldsymbol{c}''_1 = \sum_{i=0}^{l-1} d_2^i \cdot ksk_1^i \in \mathcal{R}_{PQ_l,N}$. This is followed by $\mathtt{c} = \big((d_0, d_1) + \mathtt{CKKS.ModDown}(\mathtt{c}'')\big) \in \mathcal{R}^2_{Q_l,N}$. $\mathtt{CKKS.ModDown}()$ scales down the modulus (PQ_l to Q_l), and is described in Algorithm 5 following the works [HK19, KPP22]. A more detailed and generalized description is provided in Algorithm 10 (Appendix A).

5. `CKKS.Bootstrap`: It refreshes a noisy ciphertext [BMTH21, CCS19, CHK+18a] by producing a new ciphertext with a higher depth or lower noise. As bootstrapping itself consumes a certain number of depths, the depth of a bootstrapped ciphertext, say L_{eff}, is smaller than the initial depth L after fresh encryption. Bootstrapping is required to refresh the processed ciphertexts in complex applications, such as DNN. It consists of the following four major steps.

   - **Slot to Coefficient Conversion:** Converts the ciphertext from slot form to polynomial form using a homomorphic DFT (Discrete Fourier Transformation). This involves computing a homomorphic matrix-vector multiplication, where the matrix is not encrypted, and the ciphertext is the vector.
   - **Modulus Raising:** Raises the modulus from q_0 to Q, introducing an error term that needs removal. This requires a plain ModUp operation,
   - **Coefficient to Slot Conversion:** Converts the ciphertext back to slot form after modulus raising via homomorphic IDFT (Inverse DFT) computation. This also involves computing a homomorphic matrix-vector multiplication.
   - **Homomorphic Modular Reduction:** This step applies a Chebyshev polynomial approximation to remove the introduced error term after Modulus raising. It results in refreshed computational depth.

Algorithm 4 `CKKS.KeySwitch` [HK19, KPP22] (for $dnum = L + 1$)

In: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{ksk}}_0 \in R^l_{PQ_l}, \tilde{\mathtt{ksk}}_1 \in R^l_{PQ_l}$
Out: $\mathtt{d}' = (\tilde{\mathbf{d}}'_0, \tilde{\mathbf{d}}'_1) \in R^2_{Q_l}$

for $j = 0$ to l **do**
  $d_2[j] \leftarrow \mathtt{INTT}(\tilde{d}_2[j]) \in \mathbb{Z}_{q_j}$
end for
for $j = 0$ to $l + 1$ **do**
  $(\tilde{\mathbf{c}}''_0[j], \tilde{\mathbf{c}}''_1[j]) \leftarrow 0$
  for $i = 0$ to l **do**
    $\tilde{r} \leftarrow \mathtt{NTT}(\left[d_2[i]\right]_{q_j}) \in \mathbb{Z}_{q_j}$
    $\tilde{c}''_0[j] \leftarrow \left[\tilde{c}''_0[j] + \tilde{\mathtt{ksk}}_0[i][j] \cdot \tilde{r}\right]_{q_j}$, $\tilde{c}''_1[j] \leftarrow \left[\tilde{c}''_1[j] + \tilde{\mathtt{ksk}}_1[i][j] \cdot \tilde{r}\right]_{q_j}$
  end for
end for
$\tilde{\mathbf{d}}'_0 \leftarrow \tilde{\mathbf{d}}_0 + \mathtt{CKKS.ModDown}(\tilde{\mathbf{c}}''_0)$, $\tilde{\mathbf{d}}'_1 \leftarrow \tilde{\mathbf{d}}_1 + \mathtt{CKKS.ModDown}(\tilde{\mathbf{c}}''_1)$

Algorithm 5 `CKKS.ModDown` [CHK+18b]

In: $\tilde{\mathbf{d}} \in R_{PQ_l}$
Out: $\tilde{\mathbf{d}}' \in R_{Q_l}$

$t \leftarrow \mathtt{INTT}(\tilde{d}[l + 1])$
for $i = 0$ to l **do**
  $\tilde{t} \leftarrow \mathtt{NTT}(\left[t\right]_{q_i} \in \mathbb{Z}_{q_i}$
  $\tilde{d}'[i] \leftarrow \left[p_0^{-1} \cdot (\tilde{d}[i] - \tilde{t})\right]_{q_i}$
end for

2.2 FHE Hardware design goals

A tiered structure exists in the CKKS scheme routines. The high-level or *macro* rotuines are `CKKS.Add`, `CKKS.Mult`, `CKKS.Rotate`, and `CKKS.KeySwitch`. These macro procedures apply micro procedures, such as forward and inverse Number Theoretic Transforms (NTT/INTT), dyadic Multiplication/Addition/Subtraction (MAS), and Automorphism (AUT). The NTT

is used for multiplying two N coefficients long polynomials in $\mathcal{O}(N \log N)$ time complexity, which is the asymptotically fastest one.

The special `CKKS.Bootstrap` procedure uses these macro procedures in a specific sequence to refresh noisy ciphertexts. Note that contrary to schemes like FHEW, TFHE [CGGI20] where bootstrapping is a standalone procedure, CKKS-Bootstrapping is a high-level routine which utilizes KeySwitches, Automorphisms, and MACs. Therefore, while TFHE/FHEW accelerators (e.g., [BDTV23]) focus on optimizing the programmable bootstrapping, acceleration for CKKS relies on optimized KeySwitching, Automorphisms, and MACs. Among these operations, MAC is a straightforward linear operation. Automorphism involves permutation, followed by KeySwitch [HK19, KPP22]. This permutation, if naively implemented, can become complex and expensive as the input polynomial has N=2^{16} coefficients and offers N/2 different permutations. In this work, we show how we design the permutation unit that is not only cheap in terms of area but also has linear time complexity for all permutations. The final operation- KeySwitch, is the most expensive due to the expensive ModUp step (Figure 2). Since KeySwitch is the most expensive operation, the task and data distribution approach aims to optimize this particular operation. For simplicity, KeySwitch for $dnum = L + 1$ ($K = 1$) [KPP22, HK19] is utilized throughout the paper.

2.3 Monolithic vs Chiplet packaging

In the context of large Integrated Circuits, authors in [Gon21, YLK+18, MWW+22] discuss the advantages of chiplet-based designs over monolithic designs. The problem with monolithic designs stems from the fact that to keep up with the increasing demand for high performance and functionality, chips need to be scaled up, and advanced technology nodes must be utilized. Manufacturing such big chips reduces the wafer yield as more surface area is exposed to defects per chip and increases the development cost. Such huge designs take a long time-to-market, and it is not straightforward to test and verify them. Factors such as size limitation and sub-optimal die performance due to overload contribute to a shift to SiP [Gon21].

In SiP, multiple heterogeneous smaller chiplets can be manufactured separately and later integrated together using various packaging techniques. This promotes chiplet-reuse, lowering the development costs. The chiplet-packaging techniques can be broadly classified into three main categories: 2D, 2.5D, and 3D [Gon21, MWW+22]. In 2D packaging, different dies are mounted on a substrate, known as a multi-chip module. Due to substrate limitations, this results in slow die-to-die communication and high power consumption.

To address these limitations, silicon interposers are used, and this technique is known as 2.5D integration [UCI23, PZM+23]. In this approach, an interposer is placed between the die and the substrate, enabling die-to-die connections on the interposer itself. The use of an interposer significantly enhances interconnectivity, leading to improved performance. Taking the integration capabilities a step further, 3D packaging involves stacking different dies on top of each other, akin to a skyscraper. In 3D packaging, the dies are interconnected using through-silicon vias (TSVs). 3DIC is gaining significant popularity and serves as the foundation for advancements like High Bandwidth Memory (HBM/HBM2/HBM3) [PLC+23, CPS+23, TGF09, VGT+20]. This approach significantly reduces the critical path, resulting in higher performance, lower power consumption, and increased bandwidth. The slowdown of Moore's law finds hope in 2.5D and 3D IC.

2.4 NTT Design Techniques

The multiplication of large degree polynomials is one of the major performance bottlenecks for FHE implementations. The number theoretic transform enables fast polynomial multiplications by reducing the complexity of polynomial multiplication to $\mathcal{O}(N \cdot \log N)$,

Algorithm 6 The Cooley-Tukey NTT Algorithm [Sco17a]

In: A polynomial x with coefficients $\{x_0, \cdots, x_{N-1}\}$ where $x_i \in \mathbb{Z}_q$, N, q
In: Table of $2N^{th}$ roots of unity $\boldsymbol{g}$, in bit reversed order
Out: $\hat{x} \leftarrow \texttt{NTT}(x)$, $\hat{x}_i \in \mathbb{Z}_q$, in bit-reversed order

1: $t, m \leftarrow (N/2), 1$
2: **while** $(m < N)$ **do**
3: $\quad k \leftarrow 0$
4: $\quad$ **for** $(i = 0; i < m; i = i + 1)$ **do**
5: $\quad\quad$ **for** $(j = k; j < (k + l); j = j + 1)$ **do**
6: $\quad\quad\quad V \leftarrow x[j + t] \times \boldsymbol{g}[m + i] \pmod q$ ▷ Butterfly operation starts.
7: $\quad\quad\quad x[j + t] \leftarrow x[j] - V \pmod q$
8: $\quad\quad\quad x[j] \leftarrow x[j] + V \pmod q$ ▷ Butterfly operation ends.
9: $\quad\quad$ **end for**
10: $\quad\quad k \leftarrow k + 2t$
11: $\quad$ **end for**
12: $\quad t, m \leftarrow t/2, 2m$
13: **end while**
14: **return** x

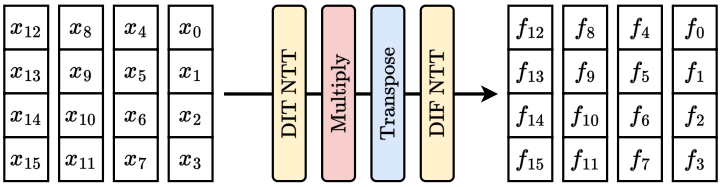

Figure 1: Hierarchical (4-step) NTT datapath for $N = 16$. DIT stands for Decimation in Time, and DIF stands for Decimation in Frequency.

and it is extensively employed for implementing FHE schemes. The NTT is defined as the Discrete Fourier Transform over $\mathbb{Z}_{q_i}$. As shown in Algorithm 6, an N-point NTT operation transforms a polynomial a of degree $N-1$ degree polynomial into another $N-1$ degree polynomial $\tilde{a}$. The NTT uses the powers of N-th root of unity ω (also referred to as twiddle factors) which satisfies $\omega^N \equiv 1 \pmod q$ and $\omega^i \neq 1 \pmod q$ $\forall i < N$, where $q \equiv 1 \pmod N$. Here, q represents an RNS base, q_i. Similarly, inverse NTT (INTT) follows the same method with the modular inverse of ω, and the resulting coefficients should be scaled by $1/N$. Prior FHE hardware acceleration works mainly utilized three techniques for implementing an NTT. The major difference between works is how they instantiate the butterfly unit highlighted in Algorithm 6.

- **Iterative.** In this type of implementation, the prior works [MAK+23, Sco17b] instantiate multiple butterfly units to process the inner 'for' loop (Algorithm 6) simultaneously, hence faster. Thus, the runtime of the NTT transformation for m butterfly units is $\sim \frac{N \cdot \log N}{2 \cdot m}$.

- **Pipelined.** This is a bandwidth-efficient implementation where the butterfly units operate in pipelines instead of parallel. It is also used by prior works [ZWZ+21, YCH22] for applications such as zero-knowledge proofs where the polynomial degree is huge. It can be considered a stage unrolled NTT, where the unrolling is done based on the outer 'for' loop instead of the inner loop (in Algorithm 6). While its latency does not decrease, its throughput is much higher and useful for applications requiring multiple polynomial NTT transformations.

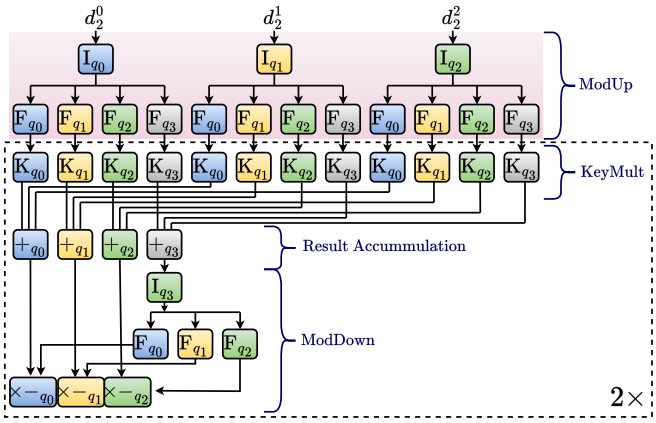

Figure 2: KeySwitch operation for $l = 2$, where I, F, and K represent INTT, NTT, and key multiplication operations using MAS, respectively.

- **Hierarchical (4-Step).** This implementation technique adopted by [FSK+21], predominantly utilizes the first technique as shown in Figure 1. It splits one polynomial into a matrix of dimensions $N_1 \times N_2 = N$. The NTT units transform N_1 coefficients using multiple butterfly units instantiated in parallel, and there are N_2 such sets to process all $N_1 \times N_2$ coefficients. After this first operation, the resultant data is transposed and multiplied with twiddle factors. Finally, other NTT units transform N_2 coefficients using multiple butterfly units instantiated in parallel, and there are N_1 such sets to process all the coefficients. To save area N_1, N_2 are chosen so that $N_1 = N_2$. This way, the same NTT unit can be utilized for both transformations ($N_1 \times N_2$, $N_2 \times N_1$). In this case, the transpose operation becomes complex and expensive as N increases.

3 FHE-tailored Multi-Chiplet Design

A widely adopted approach for realising a chiplet-based architecture is to distribute the components of one large monolithic design across multiple chiplets connected in a mesh [RKRB, KMP+21, CLSW11]. While such a disintegration approach has found utilities in applications like Machine Learning [SCV+21], they fall short in leveraging the inherent algorithmic intricacies of FHE, thereby hindering efficient workload distribution among chiplets. In the following, we investigate the trade-offs of various chiplet design possibilities for FHE and consolidate on an optimal solution. For this, let us consider the most performance-heavy macro routine- KeySwitch. Its data flow is illustrated in Figure 2 for depth $l = 2$.

(a) The naive approach for chiplet decomposition would be to closely follow the data flow of Figure 2 and allocate one chiplet per square-box in the figure (I=INTT, F=NTT, K=Key Multiplication). This approach, as shown in Figure 3 (a), will lead to (*i*) uneven allocation of chiplet resources; for example, the chiplets for NTT or INTT will be much larger than those for MAS, (*ii*) a massive increase in C2C communication overhead due to the continuous data exchange between the components, and (*iii*) increase in required on-chip memory for data duplication. Figure 3 (a) further provides a schedule for the KeySwitch operation depicted in Figure 2. In addition to illustrating the complexity of chiplet-to-chiplet communication and task dependencies, it demonstrates how most chiplets remain idle during computation, further emphasizing the imbalance in task distribution.

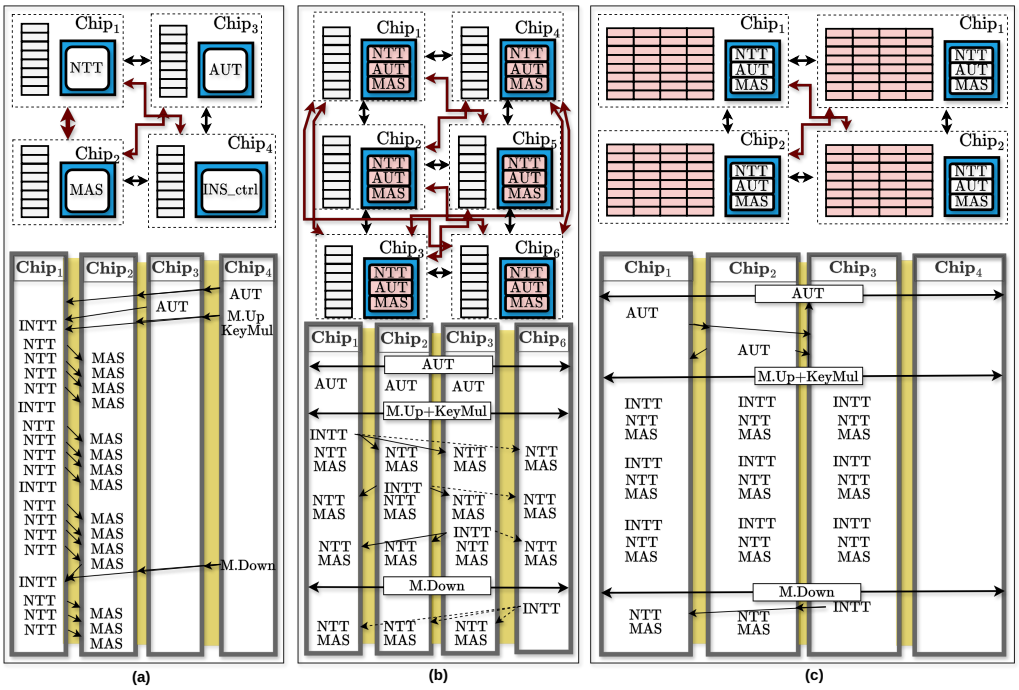

Figure 3: The diagram depicts the different techniques, data, and task distribution for automorphism followed by KeySwitch. $l = 2, 1$ for (a), (c) and 4 for (b)

For a KeySwitch operation, the communication overhead is $(l+1)\cdot(l+2)$ polynomials for ModUp and $2\cdot(l+2)$ polynomials for ModDown. The time required for the operation is equivalent to the time taken to process $(l+1)$ INTT and $(l+1)\cdot(l+2)$ NTT operations for ModUp+KeyMul and 1 INTT and $(l+1)$ NTT for ModDown of each ciphertext component. This is tabulated in Table 2 and assumes that transfer can be done fast enough (monolithic) to compute MAS operations in parallel. If the 2× slow communication cost is added, considering chiplet disintegration, the communication would take more time and surpass the computation overhead by 2×, thus becoming the major bottleneck. Hence, this approach significantly inhibits the performance of the design. A deeper disintegration would imply breaking the individual operation into several chiplets, for example, computing one NTT using several chiplets. Although this approach can lead to smaller chiplets, it will result in additional C2C communication overhead.

(b) The second approach, illustrated in Figure 3 (b), involves assigning each RNS limb computation, as depicted in Figure 2, to a single chiplet encompassing NTT, AUT, and MAS components. Computations with respect to one modulus are performed within the same chiplet. This approach also faces slow and more complex C2C communication overhead due to data dependencies between the processing of RNS limbs in the KeySwitch routine. Furthermore, as the depth of the FHE application decreases over time, reducing the number of RNS limbs (l), their respective chiplets become idle.

Figure 3 (b) illustrates the complexity of C2C communication with six chiplets, showcasing the all-to-all dependencies between the chiplets. This figure highlights that before the chiplets can begin evaluating the NTT, they must wait for the INTT operation to complete, which leads to certain chiplets sitting idle during this time. This idle time reduces the overall efficiency of the design and contributes to increased communication overhead due to the frequent data exchanges required between the chiplets. If we assume

Table 2: Comparison of the naive techniques and our technique for the KeySwitch operation, including ModUp, Key Multiplication, and ModDown. The operation count is the maximum reported by any chiplet, which will determine the overall throughput of the design. || Implies Operation is done in parallel with other computations.

	Polynomials in Communication	**Computation**			**# Chiplets**
		INTT	NTT	MAS	
Tech. (a)	$(l+3)\cdot(l+2)$	$l+3$	$(l+1)\cdot(l+4)$	$\|\|$	4
Tech. (b)	$(l+1)\cdot(l+4)$	2	$l+4$	$l+4$	$L+2$
Tech. (c)	$(l+1)\cdot(l+4)$	$l+3$	$l+4$	$l+4$	$L+2$
Our Tech.†	$4\cdot(l+3)$	$\frac{l+1}{4}+2$	$\frac{(l+1)\cdot(l+4)}{4}$	$\|\|$	4

† This technique is discussed in Section 5 for four chiplets ($r=4$).

that the result can be immediately sent to all the PU (monolithic), the runtime can be quantified as $l+1$ INTT and $2\cdot(l+1)$ NTT(+MAS) for ModUP+KeyMul, and 1 INTT and 2 NTT(+MAS) for ModDown of each ciphertext component, as tabulated in Table 2. The total data in communication for ModUp is $(l+1)\cdot(l+2)$ polynomials, as all $l+1$ INTT resultant polynomials are broadcasted to $l+2$ chiplets. Similarly, for ModDown, the communication cost is $2\cdot(l+1)$ polynomials. Hence, in a chiplet disintegration scenario where communication between chiplets is 2× slower, it becomes the main bottleneck.

(c) A refined third approach, depicted in Figure 3 (c), involves enabling each chiplet to support multiple RNS limbs to reduce C2C communication overhead and utilizing data duplication. For instance, the first chiplet can initiate NTT_{q_1} after executing INTT_{q_0} without sending it to the second chiplet, as depicted in Figure 2. This strategy minimizes communication overhead during ModUp and increases intermediate storage requirements by storing copies of input RNS limbs d_2^0, d_2^1, d_2^2 in Figure 2 on each chiplet. INTT_{q_0}, INTT_{q_1}, NTT_{q_0}, etc., are computed within the same chiplet. As shown in Figure 3 (c), this technique does not completely eliminate the complexity of C2C interconnects, as some inter-chiplet communication remains necessary to ensure duplication. Additionally, this approach requires significantly more on-chip memory since data must be duplicated across chiplets to enable local processing of multiple tasks, further increasing resource demands. This technique reduces wait time during ModUp but requires an all-to-all broadcast at the end of the computation. Hence, requiring transfer of $(l+1)\cdot(l+4)$ polynomials, including broadcast required for ModDown.

Thus, all the available approaches offer certain trade-offs and highlight communication as the major bottleneck, and our aim is to determine the best and most practical solution. With this aim, we develop a chiplet-based design approach for REED.

3.1 REED 2.5D Architecture

In a chiplet-oriented design process, there are two critical choices: the size of chiplets and the number of chiplets. The manufacturing cost is reduced, and yield increases when the chiplets are small in area. However, having many small chiplets reduces performance as the complexity of slow C2C communication increases. In this work, we will develop a chiplet design strategy and integration topology, considering the data flow of FHE. This approach provides a balance between yield, manufacturing cost, and C2C communication overhead.

As an example, Figure 4 shows a four chiplet-based REED 2.5D architecture, where the chiplets are connected in a ring formation and have exclusive read/write access to HBM in its proximity. Later, we will show that this architecture scales well with an increasing number of chiplets (Section 6.4). To overcome C2C communication overhead

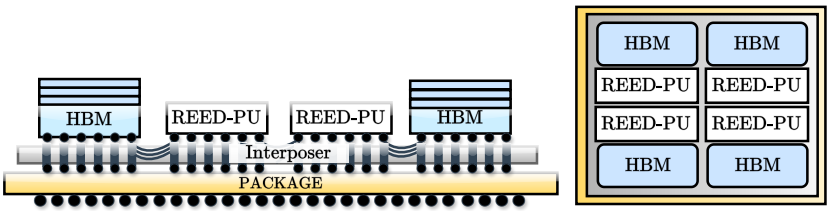

Figure 4: Side and top view of proposed four chiplet-based REED 2.5D.

and memory storage issues, we propose an RNS polynomial (limb)-oriented task and data distribution strategy, which is built on top of the third approach. Specifically, chiplets are assigned certain RNS limbs and all tasks related to these limbs without requiring data duplication (detailed in Section 5). The proposed ring formation (Section 5.4) allows us to increase the number of chiplets at the cost of only a linear increase in the number of interconnects. Hence, we can scale it to eight or sixteen chiplets as well. This ring formation for connecting the FHE chiplets is specifically tailored to the data-flow of performance-critical FHE workloads. With this formation, not all dies need to communicate with every other die simultaneously, which is crucial for minimizing C2C communication requirements.

Furthermore, our communication protocol ensures (Section 5.4) that no HBM-to-HBM communication is required. Hence, HBMs are positioned on the outer side. We also avoid sharing one HBM among multiple chiplets, ensuring that each HBM is located only in proximity to the one chiplet it serves. Notably, our placement strategy aligns well with [PZM+23, ZSB21], where authors design chiplet-based general-purpose processors with an actual tapeout, demonstrating practical viability. Finally, we also ensure a homogeneous design where all chiplets are identical, simplifying post-silicon realization.

Disintegration Granularity: Chiplet systems face a trade-off between development cost and performance degradation, depending on the disintegration granularity. Existing works on chiplet-based architectures, such as [VGT+20, TGF09, YLK+18, ZSB21, MWW+22, GPG23], show that disintegration improves yield, but it introduces challenges such as floorplanning and post-silicon testing overhead. Hence, the question:

How much disintegration is too much disintegration?

Considering a maximum die area of 800mm^2, dividing it into four chiplets offers an ≈80% yield, while eight chiplets provide a yield of ≈90%. The yield numbers are obtained from [MWW+22].

While the eight-chiplet option shows promise for achieving high yield, it faces the challenge of underutilization over time as the number of RNS limbs decreases after rescaling. Specifically, when l becomes smaller than 8, certain REED chiplets remain idle (Detailed in Section 5.3 and Section 6.4). On the contrary, the instantiation of four chiplets strikes an optimal balance between manufacturing cost and utilization. We want to remark that the number of REED chiplets is flexible and can be changed as per user requirements, depending on the technology and computation constraints.

4 Architecture Design of One Chiplet

The need for scalability and high throughput drives our design methodology. We introduce the REED design configuration- (N_1, N_2) for polynomial degree N, where $N_1 \cdot N_2 = N$. For the clock frequency f, this configuration provides a throughput of $\frac{f}{N_1}$ operations per second and can process N_2 coefficients in parallel. A configuration-flexible design approach will help obtain computation-communication parallelism within every chiplet by ensuring

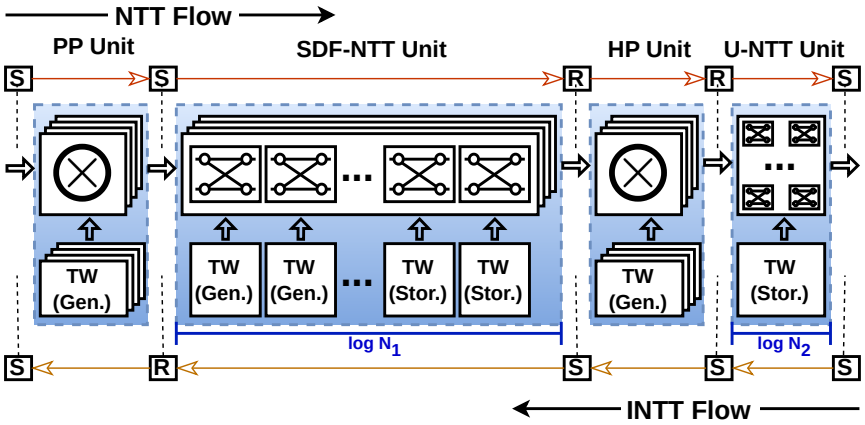

Figure 5: Novel routing-friendly Hybrid NTT/INTT design flow for $N = N_1 \times N_2$.

that the memory read/write throughput is the same as the computational throughput. Now, let us explore how we design the ingredients of REED Processing Unit (PU) to ensure flexibility.

4.1 The Hybrid NTT (Frankenstein's Approach)

The NTT/INTT unit plays a vital role in converting polynomials from slot to coefficient representation and vice versa. It is the most computationally expensive micro building-block and occupies over 50% architectural area. Therefore, designing an efficient NTT/INTT unit is crucial as it directly impacts the overall throughput and area consumption of REED.

Prior works: There are various approaches in the literature to implement NTT in hardware for large-degree polynomials, such as iterative [MAK+23, Sco17b], pipelined [ZWZ+21, YCH22] and hierarchical [FSK+21], thoroughly discussed in Section 2. The implementation complexity of the plain iterative approach increases significantly with the number of processing elements. The pipelined approach (also known as single-path delay feedback (SDF)) provides a bandwidth-efficient solution but a diminished performance. The hierarchical approach (also referred to as four-step NTT), utilized in [FSK+21], treats a polynomial of size N as an $N = N_1 \times N_2$ matrix and divides a large NTT into smaller parts. It involves performing N_1-point NTTs on the N_2 columns of the matrix, then multiplying each coefficient by $\omega^{i \cdot j}$ (where i and j are matrix row and column indices), transposing the matrix, and finally performing N_2-point NTTs on the N_1 columns.

Transposing a matrix of size $N_1 \times N_2$ requires N_1 separate memories and large data re-ordering units. Hence, in [FSK+21], the transpose unit consumes 14% of the area per compute cluster. Moreover, it also requires additional N_2 cycles for writing data to the transpose memory and N_1 cycles for reading it. Therefore, although the hierarchical approach simplifies the NTT implementation, we observe that it has the following limitations: (i) the costly transpose operation, (ii) fixed N_1 and N_2 such that $N_1 = N_2$ [FSK+21, GBP+23], and (iii) the reliance on scratchpad in some works leads to large memory fan-in and fan-out, causing routing inefficiencies.

Our Technique: We address these problems and propose a novel, scalable, and efficient NTT/INTT architecture. Our proposed design methodology-Hybrid NTT/INTT divides large NTTs into smaller parts *that may differ in size*. To eliminate the transpose operation and reduce implementation complexity, we use a different design approach that amalgamates the two approaches mentioned above, thus the name *Hybrid* NTT/INTT. A transpose is

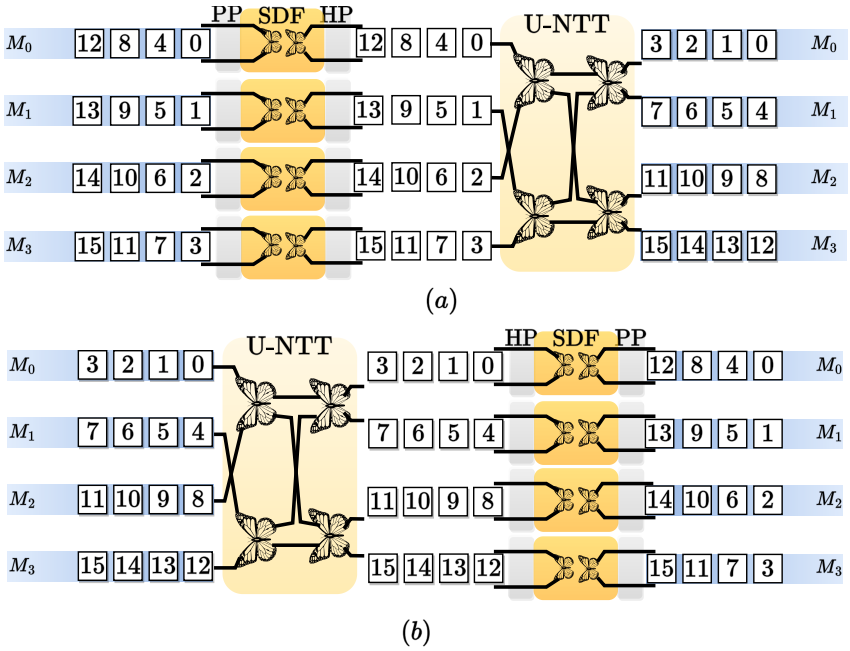

Figure 6: The proposed novel Hybrid NTT/INTT design flow with Memory access for (a) NTT and (b) INTT with $N_1 = 4$, $N_2 = 4$, and $N = 16$. The butterflies represent the Gentleman-Sande butterfly [Sco17b] operation employed in our design.

required in the hierarchical implementation approach to feed the data from unrolled NTT units back to the same unrolled NTT units. This is also the reason they need to have the same configuration ($N_1 = N_2$). If, instead, we use pipelined NTT to replace one of the unrolled NTTs, no transpose will be required. This also results in design flexibility regarding N_1 and N_2 values.

We utilize this idea to design an NTT that is routing-friendly, throughput-oriented, and does not necessitate costly transposition. To achieve this, we utilize parts of hierarchical, iterative, pipelined, and plain unrolled NTTs (Frankenstein's approach) and introduce a novel Hybrid NTT. It is fully pipelined, and its flow is shown in Algorithm 7 and Figure 5. The input polynomial for NTT operation is stored in N_2 memories of depth N_1, hence forming a matrix of size $N_1 \times N_2$ in row-major order. The NTT unit is fully pipelined and reads N_2 coefficients from memories in each cycle. We illustrated the memory layout and read the pattern of a polynomial during NTT operation for $N = 4 \times 4$ in Figure 6. NTT unit first performs pre-processing (Step 3 of Algorithm 7) using N_2 modular multipliers (PP unit), where each coefficient is multiplied with the corresponding twiddle factor.

The resulting N_2 coefficients represent one coefficient from each column of the input matrix, which should be processed via the N_1-pt NTT operation as shown in Step 6 of Algorithm 7. To avoid in-between memory communication and reduce performance, we instantiate N_2 N_1-pt single-delay feedback (SDF)-based NTT units to perform N_2 NTT operations in parallel. Each SDF-based NTT architecture consumes and generates one coefficient per cycle after filling the pipeline. It utilizes $\log_2 N_1$ cascaded butterfly units where each butterfly is coupled with a FIFO for data re-ordering [ZWZ+21].

After the SDF-NTT unit, the resulting coefficients are multiplied with powers of ω as shown in Step 9 of Algorithm 7. Similar to the PP unit, we use N_2 modular multipliers to perform this Hadamard product (HP unit). After this step, the hierarchical approach would require a transpose operation. To eliminate the transpose, we employ one fully unrolled N_2-pt NTT architecture (N-NTT unit) that takes N_2 coefficients as input per

Algorithm 7 Hybrid NTT with NWC

In: a (a matrix of size $N_1 \times N_2$ in row-major order)
In: ω (N-th root of unity), ψ ($2N$-th root of unity)
Out: $\tilde{a} = \texttt{NTT}(a)$ (a matrix of size $N_1 \times N_2$ in column-major order)

for $(i = 0; i < N_1; i = i + 1)$ **do**
  for $(j = 0; j < N_2; j = j + 1)$ **do**
    $a[i][j] \leftarrow a[i][j] \cdot \psi^{i \cdot N_2 + j} \pmod q$ ▷ PP:Pre-processing
  end for
end for
Apply N_1-pt NTT to the columns of a ▷ using SDF-NTT
for $(i = 0; i < N_1; i = i + 1)$ **do**
  for $(j = 0; j < N_2; j = j + 1)$ **do**
    $a[i][j] \leftarrow a[i][j] \cdot \omega^{i \cdot j} \pmod q$ ▷ HP:Hadamard prod.
  end for
end for
Apply N_2-pt NTT to the rows of a ▷ Unrolled(U)-NTT
return a

cycle and generates N_2 coefficients as output per cycle while performing a cost-free natural transpose operation. It does not incur an extra run-time cost as it is merged with the final step (N_1 N_2-pt NTTs). Thus, the proposed Hybrid design reads N_2 coefficients per cycle and generates N_2 coefficients per cycle after filling the pipeline. It is scalable and enables different area/performance configurations by changing N_1 and N_2 values. This unit features bi-directional flow, and based on the configuration parameters, it provides a throughput of $\frac{f}{N_1}$. Overall, the properties and advantages of the proposed unit are as follows.

- **Transpose elimination:** The Hybrid NTT eliminates transpose by using two orthogonal NTT approaches, pipelined (SDF) approach for N_1-sized NTTs and unrolled (U-NTT) approach for N_2-sized NTTs. As shown in Figure 6 (a), the output coefficients of SDF-NTT are processed directly by U-NTT, providing a seamless, natural transpose operation.

- **Bi-directional workflow:** The above method of transpose elimination also helps make our NTT unit bi-directional (for INTT), as illustrated in Figure 6 (b). The additional routing complexity is balanced with efficient pipelining.

- **Low-level optimizations:** For modular multiplication and reduction unit, we adopted the word-level Montgomery [Mon85, MÖS19] modular reduction algorithm. and optimized it for our special prime form, $2^{w-1} + q_H \cdot 2^m + 1$, where $m = 18$ is Montgomery reduction size, and $\lceil \log_2 q_H \rceil = 10$ is small. A total of $(N_2+1)\log_2(N1)+ \frac{N_2}{2}(\log_2(\frac{N_2}{2}) + 5) - 7$ modular multipliers are utilized.

- **On-the-fly twiddle generation:** We employ commonly used on-the-fly twiddle generation with a small constant memory that stores a few initial roots of the unity. This helps reduce the on-chip constant storage by up to 98.3%.

One REED-NTT requires $((N_2+1)\cdot\log_2(N_1)+(\frac{N_2}{2})\cdot\log_2(\frac{N_2}{2})+(5\cdot\frac{N_2}{2})-7)$ multipliers. The proposed transpose-less NTT also finds applications in other cryptographic schemes. For example, in Zero-Knowledge proofs [LWY+23], where higher polynomial degrees make transposing more expensive, our NTT design offers a better area-time tradeoff.

Algorithm 8 Automorphism

In: $a[N_1][N_2], gle$
Out: $\hat{a} = \rho(a)$

1: $index \leftarrow gle$
2: **for** $(l_0 = 0; l_0 < N_1; l_0 = l_0 + 1)$ **do**
3: $\quad l_1 \leftarrow index \pmod{\log(N_1)}$
4: $\quad start \leftarrow index \gg \log(N_1)$
5: $\quad addr[j] \leftarrow (start + j \cdot gle) \pmod{\log(N_2)} \; \forall \; j \in [0, N_2)$
6: $\quad \hat{a}[l_1] \leftarrow \texttt{shuffle\_tree\_}N_2 \times N_2(addr, a[l_0])$
7: $\quad index \leftarrow index + gle$
8: **end for**
9: **return** $\hat{a}[N_1][N_2]$

4.2 Multiply-Add-Subtract (MAS) and Automorphism (AUT)

MAS is elementarily designed as a *triadic* unit for computing point-wise multiplication, addition, subtraction, or multiply-and-accumulate operations. It utilizes the modular multiplier proposed for the Hybrid NTT unit. A high-level overview of the MAS unit is shown in Figure 7. On the other hand, designing an efficient AUT unit is challenging [FSK+21, SFK+22]. It permutes ciphertexts using the Galois element (gle) to achieve rotation or conjugation. A polynomial is stored as a matrix $N_1 \times N_2$ in N_2 memories.

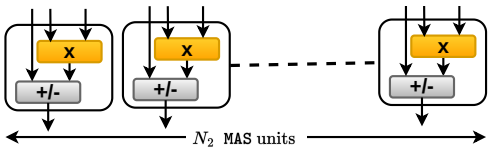

Figure 7: The set of triadic MAS units.

For AUT, we make a key observation that when we load N_2 coefficients from memory address l_0 across all N_2 memories, they are shuffled based on the desired rotation offset (ρ_{rot}), and then written to address l_1 across all N_2 memories. Hence, even though the coefficient order is shuffled, they all go to the same address of N_2 distinct memories. We utilize this property to permute all N_2 coefficients in parallel. This out-of-place automorphism is presented in Algorithm 8. The in-place permutation techniques proposed in previous works [FSK+21, SFK+22] increase routing complexity due to memory transposition requirements. We are still left with a quadratically complex and expensive shuffle $\mathcal{O}(N_2^2)$ among the coefficients. However, we analyzed that all the shuffles could be performed pairwise on the coefficient batches, as shown in Figure 8. After each stage, two batches of coefficients are merged to form a new batch. We replace the naive and expensive operation with a pipelined binary-tree-like shuffle. Its number of pipeline stages adjusts with N_2, making the pipelining scalable and efficient for higher configurations. Moreover, the unit can handle any arbitrary rotation.

4.3 PRNG-Based Partial Key-Switching Key Generation

A PRNG is deployed to generate half the key components on the fly. The idea here is that the $ksk_1 = a$ component of the key is generated by expanding a public seed and is assumed to be in NTT form, meaning it does not undergo an NTT transformation. Therefore, instead of storing the complete polynomial, the cloud or server can store the seeds of the key and generate them on the fly when needed. Note that the polynomials are

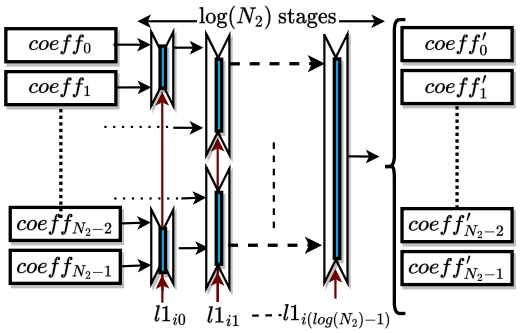

Figure 8: An example of a `shuffle_tree_`$N_2 \times N_2$ workflow. Every stage has sufficient registers to hold N_2 coefficients.

generated in RNS form and have no interdependency. In other words, for ciphertext at depth, l, the $(l+1) \cdot (l+2)$ seeds are expanded, and the remaining limbs are ignored.

Our implementation uses the Trivium unit for PRNG similar to [MAK+23], which takes a 64-bit seed as input and initializes over 18 clock cycles, performing 18 rounds of operations. After initialization, the Trivium core generates 64-bit pseudorandom data continuously, producing one word per cycle without stalling. Each PU is equipped with one Trivium unit. The keys are only used for Key Multiplication during KeySwitch and are directly fed to the MAS unit, which simultaneously consumes the pseudorandom words generated by the Trivium cores. The on-the-fly generation of the ksk_1 substantially reduces the memory footprint for key-switching. By generating it during runtime, we cut the required bandwidth (HBM-to-chiplet) and storage space for the key-switching key by 50%. This dynamic approach optimizes memory usage and increases the efficiency of the system for key-switching operations. With this, we conclude the design of micro procedures.

4.4 Programmable Instruction-Set Architecture

In this section, we discuss how to program the micro procedures (NTT, AUT, MAS) for high-level FHE routines. This is crucial for determining their placement in the architecture, which will be discussed in the subsequent section.

Prior works define a strict operation flow. This prevents adaptations to future changes in the FHE algorithms or routine flow. Noting this, we utilize an instruction-based architecture design technique [SRTJ+19], wherein a relatively small instruction controller programs the REED-PU, manages the multiplexers, and collects 'done' signals from these units. The instruction controller includes a small BRAM for storing instructions. These instructions are provided at the start of the computation, along with the input data. Once the user sends an execute command, the instruction controller manages the execution of all stored instructions. This setup gives users the flexibility to specify any desired set of instructions, such as only performing NTT processing, and allows customization of parameters. Two types of instructions handled by the instruction controller are *micro* and *macro*. Micro-instructions are low-level arithmetic procedures like NTT, INTT, point-wise modular addition, subtraction, multiplication, multiplication-and-accumulation, and automorphism. They are used to compose microcodes for realizing macro-instructions for homomorphic addition/multiplication, KeySwitch, rotation, and moddown. A bootstrapping is performed using these macro instructions.

We initiate the PU design, as shown in Figure 9, and uncover two important design decisions. The first is regarding the placement of NTT/INTT and MAS/AUT units. The second deals with the problems associated with large on-chip memories utilised in prior works [KLK+22, KKC+23] for storing keys.

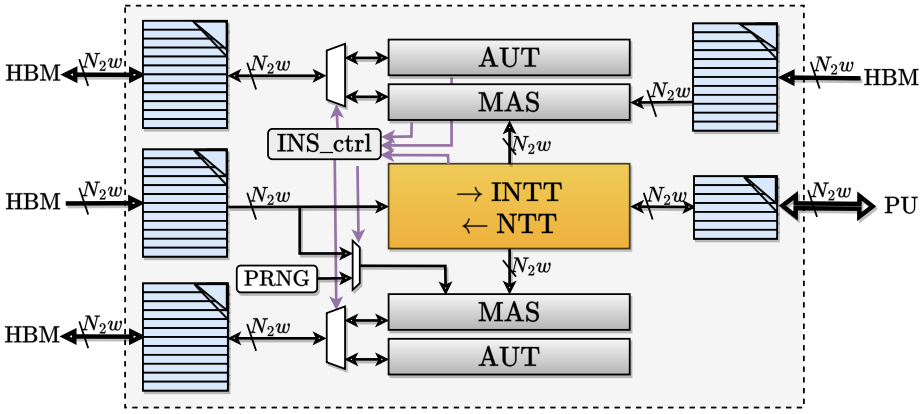

Figure 9: The REED-PU design. Every data communication (memory to building blocks and off-chip to on-chip) here has a bandwidth of N_2w bits/clock-cycle.

4.5 REED Processing Unit (PU)

We initiate the PU design, as shown in Figure 9. A PRNG is deployed to generate half the key components on the fly [MAK+23]. We uncover two important design decisions. The first is regarding the placement of NTT/INTT and MAS/AUT units. The second deals with the problems associated with large on-chip memories utilised in prior works [KLK+22, KKC+23] for storing keys.

❶ The polynomial processed by NTT unit is multiplied with two polynomials of KeySwitch keys and accumulated. Hence, we instantiate a pair of MAS/AUT units capable of simultaneously processing both key components. Thus, the design has the ability to run NTT and MAS units concurrently (shown in Figure 10), which improves the KeySwitch performance by 66.7%, as explained in Section 4.6. Moreover, since the AUT and MAS units are relatively cheaper, this design decision also does not add significant area overhead, as presented later in Table 4.

❷ In hardware accelerators, on-chip memory causes significant area overhead. Chiplets with large on-chip memory are not power, area, and manufacturing cost efficient [Sie14]. In the context of FHE, each KeySwitch key demands ≈1MB or 91MB storage for $dnum = L + 1$, and $L = 30$ or $dnum = 3$, and $L = 22$ respectively. Given the limited on-chip memory capacity of small chiplets, accommodating even a single KeySwitch key becomes rather challenging. Consequently, reliance on off-chip memory access becomes essential when a KeySwitch operation necessitates a different key. It is also not useful to store just one relinearization key required after ciphertext multiplication, as for a majority of applications, more rotations are required compared to multiplications [JKLS18]. Therefore, in our architecture, we store all the keys in the large off-chip HBM. To reduce the overhead of off-chip memory access, we develop an efficient prefetch unit that streamlines data movements in parallel to computation, as described next.

4.6 Throughput Computation for KeySwitch

The KeySwitch, detailed in Algorithm 4, is the most expensive operation among all the routines. For $dnum = L + 1$, it transforms all $L + 1$ residue polynomials from slot to coefficient representation (INTT), and then each of these is transformed to $L + 2$ NTTs, multiplied with two key components, and accumulated. This requires $L + 1$ INTTs, $(L + 1)(L + 2)$ NTTs, and $2(L + 1)(L + 2)$ MAS, making the naive throughput of this operation: $\frac{f}{(L+1)(1+3(L+2))\cdot N_1}$. By utilizing REED's parallel processing capability to

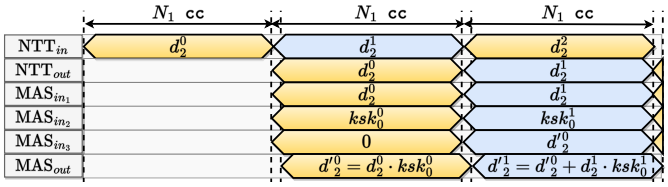

Figure 10: Timeline of parallel and pipelined operation flow.

perform all MAS operations concurrent to the NTT operations (shown in Figure 10), we save $2(L+1)(L+2)$ clock cycles and increase the throughput to $\frac{f}{(L+1)(L+3)\cdot N_1}$, resulting in a 66.7% improvement.

4.7 Streamlined Prefetch for On-Chip Storage

As mentioned in Section 4.1, REED's design methodology mitigates the need for scratchpad-like on-chip memory, allowing us to use memory units solely as prefetch units.

Each memory unit in Figure 9 exhibits balanced fan-in and fan-out, and among the five memory units depicted, four are fed by off-chip memory. The small memory is responsible for storing and communicating the INTT result to the other PUs (or Chiplets) (elaborated in Section 5.4). Only two of the four memories communicating with off-chip memory need to write back the results, as illustrated by bi-directional arrows in Figure 9.

Summarily, three memories perform off-chip read/write communication. These memories are physically divided into two parts. When one is utilized for on-chip computation, the other performs off-chip prefetch (similar to ping-pong caching).

5 Techniques for exploiting Comm-Comp Parallelism

The previous section discussed how a configuration-based design methodology enables off-chip and on-chip communication-computation parallelism. However, when distributing FHE workloads among multiple chiplets, we must consider the C2C communication overhead. This is important as state-of-the-art HBM3 [Ram] features a bandwidth of 1.2TB/s, while the state-of-the-art C2C interconnect UCIe [UCI23, Syn23] only offers a bandwidth of 0.63 TB/s. Consequently, a chiplet system optimized for HBM bandwidth could face bottlenecks due to slow C2C communication. The routine that necessitates most data exchange is the ModUp portion of KeySwitch, detailed in Algorithm 9. It switches the modulus of each $L+1$ residue polynomial ($\texttt{INTT}(d_{2q_i})_{q_i} \ \ \forall\, i \in [0, L]$) to $(L+2)$ residues polynomials ($\texttt{NTT}(d_{2q_i})_{q_j} \ \forall\, j \in [0, L+1]$). The ModUp operation is followed by key multiplication and ModDown. This is the flow for $dnum = L+1$, and a more detailed discussion of our proposed technique for $dnum < L+1$ is provided in Appendix A.

5.1 Communication cost-analysis

In a multi-PU work [MAK+23], the authors briefly discuss limb-based decomposition and propose distributing computation across the RNS polynomials (limbs) by employing one PU per limb. While this approach enables highly parallel computations, as the multiplicative depth decreases, many PUs become idle, causing underutilization.

In [KKK+22, KLK+22, KKC+23], the authors utilize both the limb-based and coefficient-based task distribution during KeySwitch. They [KLK+22, KKC+23] propose using limb-wise distribution for INTT and NTT steps and coefficient-wise distribution for the modulus switch (referred to as Base Conversion in the paper). [KLK+22] utilizes four PUs, and since thus base conversion is needed between the INTT and NTT steps, all-to-all broadcasts are

done across PUs to switch from one distribution to another. In the subsequent subsection, we analyse how both techniques attain the same communication overhead.

Additionally, the all-to-all C2C broadcast between the chiplets is slow and increases by $\mathcal{O}(\mathtt{r}^2)$ with the number of chiplets $\mathtt{r}$. In the context of FHE, switching between limb- and coefficient-wise task distribution becomes expensive as it demands all-to-all C2C data movements. For example, [KLK+22] proposes utilizing four PUs in a single monolithic chip. However, when extended to a chiplet setting, where each PU occupies a separate chiplet, the lack of an all-to-all broadcast capability makes it difficult to send data across all chiplets instantly. Using bi-directional C2C communication ability, the polynomial would reach all four chiplets via at least two serial C2C communication interfaces. The on-chip bandwidth used in the prior works is (20TB/s [KLK+22], 36TB/s [KKC+23]) is much less than the state-of-the-art C2C communication bandwidth (0.63TB/s [UCI23, Syn23]). As a result, chiplets would have to wait longer for data to arrive before computing, and this C2C communication overhead will significantly inhibit the performance. Hence, there is a need to devise a schedule that can couple most of the communication with computation.

5.2 Limb-wise vs Coefficient-wise distribution

[KLK+22] utilizes four PUs and states that after the ModUp step, the number of polynomials that need to be transferred during limb-based only decomposition is $2 \cdot dnum \cdot (L+K+1)$, while for coefficient-wise it is $(dnum + 2) \cdot (L + K + 1)$. They present this discussion in the context of a generalized KeySwitch technique for an arbitrary value of $dnum$, presented in Algorithm 10 (in Appendix A).

This assumes the limb-wise distribution is done after multiplication with KeySwitch keys, and all the results are sent to every PU via an all-to-all broadcast. However, we remark that one PU does not need to send all the polynomials and instead only needs to send $\frac{2 \cdot (dnum-1) \cdot (L+K+1)}{dnum}$ polynomials so that every PU holds the results for the supported bases. Therefore, the total cost becomes $2 \cdot (dnum - 1) \cdot (L + K + 1)$, which is less than the cost of coefficient-wise distribution for $dnum = 3$. Furthermore, we would like to note that polynomial distribution after KeySwitch is expensive as the data doubles in size after multiplication with the two key components. Hence, this distribution should be done immediately after NTT computation, which further reduces the cost to only $(dnum - 1) \cdot (L + K + 1)$. This is much less compared to coefficient-wise distribution $(dnum + 2) \cdot (L + K + 1)$. Hence, we reassert that limb-based distribution will attain less communication overhead than coefficient-wise without any extra computation overhead.

5.3 Data Distribution across Multiple Chiplets

The above analysis establishes that limb-based decomposition is the best task and data distribution technique across multiple chiplets. Within one chiplet, the computation is coefficient-wise distributed as N_2 coefficients are processed in parallel (discussed in Section 4). However, as discussed in Section 3 (b) and (c), the limb-based distribution technique also has limitations. Therefore, we adapt it to offer long-term high performance.

This adaptation stems from the two key observations from the data flow of the KeySwitch operation in Figure 2. Firstly, the INTT results that need to be shared and duplicated are ephemeral, and thus, they do not require any long-term storage – subsequent operations immediately consume them. To improve the efficiency, instead of duplicating the INTT data or sharing memory across chiplets, we leverage a *ring-based C2C communication* as shown in Figure 11. The chiplets are connected in a ring formation where each chiplet processes one INTT result and then sends it to the next chiplet. The ring-based data movement minimizes the number of C2C interconnects and ensures that each chiplet operates on independent memory, simplifying placement and routing constraints.

The second observation is related to the underutilization of chiplets with reduced levels or depths in the ciphertext after homomorphic rescaling. If we distribute the limbs across the chiplets in an interleaved manner, then as the multiplication depth decreases, the number of limbs per ciphertext in each chiplet also uniformly decreases. To explain the benefits of the interleaved distribution, we will use the analogy of a card game.

The FHE card game: In this game, each player represents a chiplet, while the residue polynomials or limbs act as the cards. The cards dealt out are collected in a LIFO (last-in-first-out) manner [1], imitating the loss of multiplicative depth during computation. All players engage in the game (FHE routine computation) until they exhaust their cards. The start of a new FHE computation game mirrors Bootstrapping, which starts when only one player retains a single card. Until this point, players without cards must wait until the next game to participate. With these rules, the dealer (user or compiler [VJHH23]) has two choices: deal out all the cards to one player before moving to the next or do alternate distributions such that every other card goes to different players. Let us say there are $L = 32$ cards and $\mathtt{r} = 4$ players. In the former case, the first player gets the cards drawn at instances $\{0, 1, 2, \cdots, 7\}$, and in the latter case, at instances $\{0, 4, 8, \cdots, 28\}$, and so on for the other players.

In both scenarios, each player receives 8 cards in total. In the first option, the last player exhausts their cards first, followed by the preceding player, and so on and so forth. Consequently, until the first player runs out of cards, others remain inactive. Conversely, with alternating distribution, each player loses one card in turn. Thus, at any point in the game, players either possess the same number of cards or one less, ensuring active involvement throughout.

The goal of FHE architecture design is to ensure the full utilization of chiplets in the long term and, thus, deliver high performance for the available computation resources. It translates to maximizing the player interaction in the FHE card game. Thus, the latter technique of interleaved alternate distribution offers maximum chiplet participation in the "card game" of computations. Now, if there are too many players, then the number of players becoming idle will increase no matter which technique is used. The latter technique will only minimize the idle time. This is the problem with using more chiplets. Thus, next, we will discuss our final key technique for reducing communication overhead and then derive a good upper-bound on the number of chiplets.

5.4 Efficient Non-Blocking C2C Communication

The proposed ring-based communication still faces overhead as the chiplets have to wait for every chiplet before them in the ring to send the INTT result. Hence, we propose a communication strategy, illustrated in Figure 11, to overcome this remaining problem.

The key idea is that the chiplets concurrently operate on different limbs instead of waiting for one limb and then processing it, as shown in Algorithm 9. Each chiplet starts with the assigned limb, computes INTT, and then performs multiple NTTs on it. While performing NTT, it starts sending/receiving the INTT result. For example, the REED_0 sends its INTT result to REED_3 and receives the INTT result from REED_1. This is a uni-directional *ring-based communication*. Since only one INTT result needs to be sent for $\frac{l+2}{\mathtt{r}}$ parallel NTT computation, we have a larger C2C communication window compared to computation. Consequently, *non-blocking communication* is achieved as data computation can proceed concurrently with relatively slower communication.

This technique necessitates just one read/write port per chiplet, in contrast to the requirement for $(\mathtt{r} - 1)$ ports in a star-like (i.e., all-to-all) C2C communication network.

[1] It can also be FIFO (first-in-first-out) without any loss of generality.

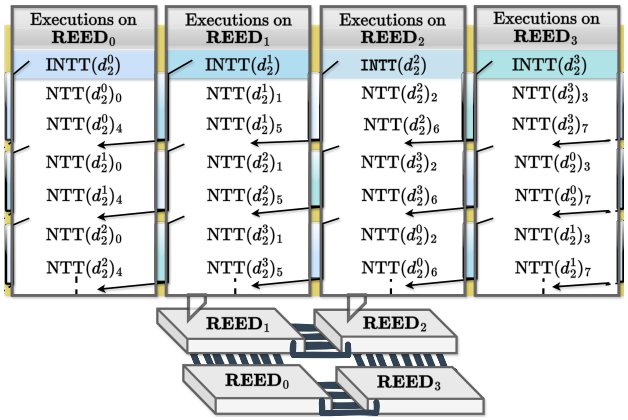

Figure 11: Non-blocking ring-based communication for four REED chiplets when $l = 6$. The blocks between executions represent the long communication window to make up for slow inter-chiplet (C2C) communication.

The Adapted Data and Task Distribution Technique.

C2C communication bandwidth plays a major role in the performance versus the number of chiplets trade-off. Let us assume the C2C communication bandwidth is k× slower than the HBM to Chiplet communication bandwidth. Each chiplet operates on $\frac{L+2}{\mathtt{r}}$ polynomials. The total computation time to process all $L+2$ polynomials for the KeySwitch should be close to k. Otherwise, we will not be able to decouple the communication from computation as discussed above. This offers us a loose upper bound $\mathtt{r} < \frac{L+2}{k}$. To ensure u× higher utilization, this bound must be made tighter. u can take any value $\leq \frac{L+2}{k}$ ($u = \frac{L+2}{k}$ for monolithic chip). We take u as 4. The adapted limb-based task/data distribution technique has the following properties:

- The number of chiplets is constrained by $\mathtt{r} \leq \frac{L+2}{4k}$.
- An interleaved data/task distribution approach is utilized such that Chiplet$_i$ gets data and task corresponding to limbs $\mathtt{r}j + i \;\forall\; 0 \leq j < \frac{L+2}{\mathtt{r}}$, instead of sequential allotment (Chiplet$_i \leftarrow \mathtt{r}i + j$).
- The technique outlined in Algorithm 9 is to be followed by all Chiplets to minimize data exchange overhead and costly C2C interconnects.

5.5 ModDown/Rescaling Task flow

So far, we have discussed the data and task flow in the context of ModUp operation, which exhibits the highest complexity in terms of NTT operations. Next, we turn our attention to the ModDown/Rescaling operation. As shown in Figure 2, the ModDown operation constitutes the final step of the KeySwitch procedure. This operation involves a single INTT followed by $l+1$ NTT operations, resulting in significantly lower complexity compared to the ModUp operation. Figure 12 (a) illustrates the communication flow for ModDown across multiple chiplets. Specifically, the chiplet that holds the polynomial corresponding to the modulus being dropped performs the INTT and sends the resulting data to the next chiplet. The data is then propagated through the remaining chiplets in a feed-forward ring topology, as depicted in the figure. Notably, the MAS operations are done simultaneously with the NTT operations.

While the overall computational complexity of the operation is $\frac{l+1}{r}$ NTTs per chiplet, there is an additional fixed communication overhead of $r-1$ polynomials due to inter-chiplet data transfer. A similar flow is observed in the Rescaling operation, which reduces

Algorithm 9 ModUp_KeyMul

In: $\boldsymbol{d_2}$ (the ciphertext component to be linearized)
Out: BUF $= ModUp\_KeyMul(\boldsymbol{d_2})$

Following tasks are executed by $\text{REED}_i\ \forall\ i \in [0, r)$
▷ All REED_i operate **in parallel** as shown in Figure 11
for $(j = 0; j < \frac{l+1}{r}; j = j + 1)$ **do**
  $\texttt{I}_\texttt{i}^{\texttt{rcv}} \leftarrow \texttt{INTT}(\boldsymbol{d}_2^{j \cdot r+i})$
  ▷ *Initiate communication with* $\text{REED}_{(i+1) \bmod 4}$, $\text{REED}_{(i-1) \bmod 4}$
  for $(m = 0; m < r; m = m + 1)$ **do**
    $\texttt{I}_\texttt{i}^{\texttt{proc}} \leftarrow \texttt{I}_\texttt{i}^{\texttt{rcv}}$
    ▷▷ *Long Communication window opens now* ◁◁
    ▷ *Receive* $\texttt{I}^{\texttt{rcv}}_{\texttt{(i+1) mod 4}}$ *from* $\text{REED}_{(i+1) \bmod 4}$ *and Send* $\texttt{I}_\texttt{i}^{\texttt{rcv}}$ *to* $\text{REED}_{(i-1) \bmod 4}$
    for $(t = 0; t < \frac{l+2}{r}; t = t + 1)$ **do**
      $\text{BUF}_{t*r+i}\ +=\ (\texttt{NTT}(\texttt{I}_\texttt{i}^{\texttt{proc}}) \cdot \texttt{KSK}^{j \cdot r+(i+m) \bmod 4})_{q_{t*r+i}}$
    end for
    ▷▷ *Ensure* $\texttt{I}^{\texttt{j·r+(i+m+1) mod 4}}$ *has been received*
    ▷▷ *Communication window closes* ◁◁
    $\texttt{I}_\texttt{i}^{\texttt{rcv}} \leftarrow \texttt{I}^{\texttt{rcv}}_{\texttt{(i+1) mod 4}}$
  end for
end for
return BUF

ciphertext depth following noise growth. However, Rescaling is not strictly needed after the KeySwitch procedure and can be performed independently to mitigate noise after several rotations, accumulations, or plaintext multiplications. ModDown and Rescaling are applied to each ciphertext component, requiring two polynomial broadcasts. Note that the figure shows the processing of one component as the second component's communication will be done in parallel with the first component's computation. The INTT will be computed by the same chiplet and transferred, while the other chiplets compute the NTT for the previous component. Hence, no communication overhead will be incurred for the second component. Runtime of ModDown and Rescaling is included in the total runtime reported for the Relinearization operation (tabulated in Table 3).

When Rescaling is required immediately after a ModDown operation, it leverages the optimized flow from the ModUp operation, as demonstrated in Figure 12 (b). Both processes necessitate an INTT broadcast followed by NTT computation, where the chiplet executing the INTT performs one less NTT at best, effectively dropping the computation with respect to the modulus used for the INTT. Consequently, while the first set of NTT computations is underway, the second INTT result can be broadcast; thus, wait time overhead is incurred only once overall.

6 Implementation Results

Based on the precision-loss study done for $dnum = L + 1$ (shown in Section 7.1), we choose the overall parameters for synthesizing and benchmarking our design as $N = 2^{16}, L = 30, K = 1, L_{boot} = 15, w = 54$. Upon implementation (silicon realisation), only the parameters N, w are fixed, and the other parameters (e.g., $dnum$) can be changed as per application requirements. For a higher value of K, the task distribution is discussed in Appendix A. The user can control this via the low-level instruction abstraction provided.

We synthesize the entire REED 2.5D PU for configurations 1024×64 and 512×128 using TSMC 28nm and ASAP7 [CVS+16] 7nm ASIC libraries with Cadence Genus 2019.11,

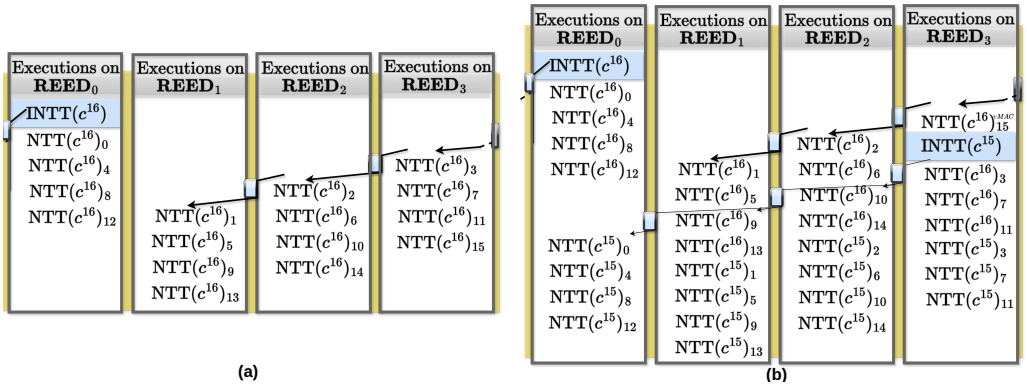

Figure 12: Ring-based communication for four REED chiplets when $l = 15 \rightarrow 16$ (after ModUp) for the (a) ModDown operation and (b) ModDown+Rescale operations.

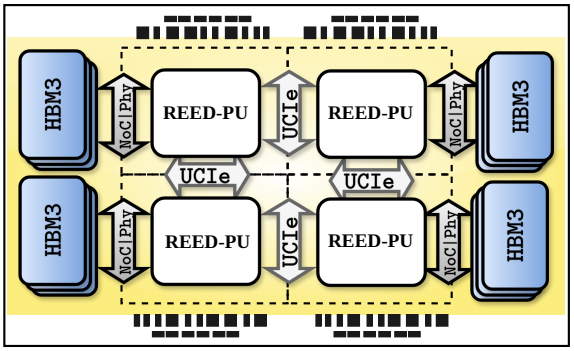

Figure 13: The complete architecture diagram of 4-chiplet REED 2.5D for 1024×64 configuration. The multiple small black blocks denote I/O interconnects along the edges.

and SRAMs are used for on-chip memories (with total storage capacity of 14 polynomials). The REED-PU and all its building blocks are fully implemented using Verilog Hardware Definition Language. We simulated our design using Vivado 2022.2 for functionality testing. The Instruction Controller is a part of the design and is connected to the master processor via the same master-slave interface as data in prior works [MAK+23]. Our primary objective is to achieve high performance while optimizing area and power consumption. To this end, we set our clock frequency target to 1.5 GHz, use High-vt cells (hvt) configuration for low leakage power, enable clock-gating, and set the optimization efforts to high. We set the input/output delays to 20% of the target clock period and leverage incremental synthesis optimization features. Moreover, we take a step further by *prototyping* the building blocks on Xilinx Alveo U250 to verify functional correctness, which has not been done by prior ASIC FHE accelerators.

As off-chip storage, we leverage the state-of-the-art HBM3 [Ram, JED22, PLC+23, LLL+15] memory, offering improved performance and reduced power. It is already deployed in commercial GPUs and CPUs [EH22]. HBM3 with 8/12 stacks of 32Gb DRAMs has 32/48 GB storage capacity [JED22, Mur23]. The ciphertexts provided by the client can be transferred to REED using 32 lanes PCIe5 offering a bandwidth of 128 GB/s [SYY+23]. In our work, we present results for HBM3 PHY and HBM3 NoC (Network On Chip), based on [Ram, PLC+23, CPS+23] with reported bandwidth of 1.2TB/s [Ram]. For C2C communication, UCIe (Universal Chiplet Interconnect Express) advanced interconnect can offer a bandwidth of 0.63 TB/s [UCI23, Syn23] for 2.5D integration. Thus, we can send

Table 3: Performance micro-benchmarks for 28nm and 7nm.

Micro-Benchmarks ↓ **Configuration →**	**Level**	**Time (ms)**	
	l	**1024×64**	**512×128**
AUT/MAS (pt-ct)	30	0.005	0.003
MAS (ct-ct)	30	0.01	0.005
KeySwitch	30→31	0.19	0.08
MULT & Relin.	30→29	0.22	0.11
Bootstrapping	1→30→15	14.2	7.1

Table 4: Total area consumption of 4-chiplet REED 2.5D for different configurations on 28nm and 7nm.

Components	**28nm (mm^2)**		**7nm (mm^2)**	
	1024×64	**512×128**	**1024×64**	**512×128**
REED	74.9	115	24	43.9
REED-PU	58.0	81.0	7.01	9.9
⌊ NTT/INTT	38.2	56.8	5.61	7.9
⌊ 2×MAS	3.1	6.6	0.42	0.76
⌊ PRNG	0.15	0.28	0.02	0.04
⌊ 2×AUT	0.14	0.32	0.02	0.04
⌊ Memory	16.1	16.1	1.2	1.2
HBM PHY/NoC	16.9	33.8	16.9	33.8
4×REED	299.6	392.4	96	175.6
C2C	12.32	14.64	0.8	1.6
Total Area	**311.9**	**461.4**	**96.7**	**177**

or receive 64, $54-bit$ coefficients per clock cycle. Overall we present the results using the following: (i) Verilog description of REED-PU, (ii) Instruction set modeling of the rest, (iii) Synthesis result in two technologies, (iv) Power simulation of a prelayout netlist for single clock, (v) Extrapolation of the power consumption of single clock cycles for the cycle accurate simulation, (vi) Power values estimated from the cycle accurate simulation, and (vii) Area estimations after synthesis.

Table 4 presents the area results for the REED 2.5D architecture, featuring a 4-chiplet configuration as illustrated in Figure 13. This design conforms to the fabricated chiplets systems [Int23, AMD23]. The inner REED-PU, NoC, and HBM (shown in Figure 13) constitute one chiplet (similar to [PZM+23, ZSB21]). In Table 3, we present the performance of FHE routines for both configurations (512×128 and 1024×64) with the achieved target clock frequency of 1.5 GHz. Section 4.6 explains how the throughput of KeySwitch is obtained. For $dnum = L + 1$, we report high (≈99.9%) utilization which reduces to ≈95% for $dnum = 3$, as detailed in Section A.1.

6.1 Power and Performance Modelling

Using a cycle-accurate model, we obtain the performance and power consumption estimates for REED. REED's communication and computation are decoupled by design and do not need application-specific schedules to reduce data load/store stalls. REED also has a modular architecture design; all chiplets are identical, and none of the elementary building blocks is distributed across chiplets. The REED-PU was fully described using Verilog Hardware Definition Language. Its complete functionality is tested via simulations in Vivado 2022.2. For the results, one REED-PU was fully synthesized to obtain overall area results using Cadence Genus and TSMC 28nm libraries. The entire system was modelled

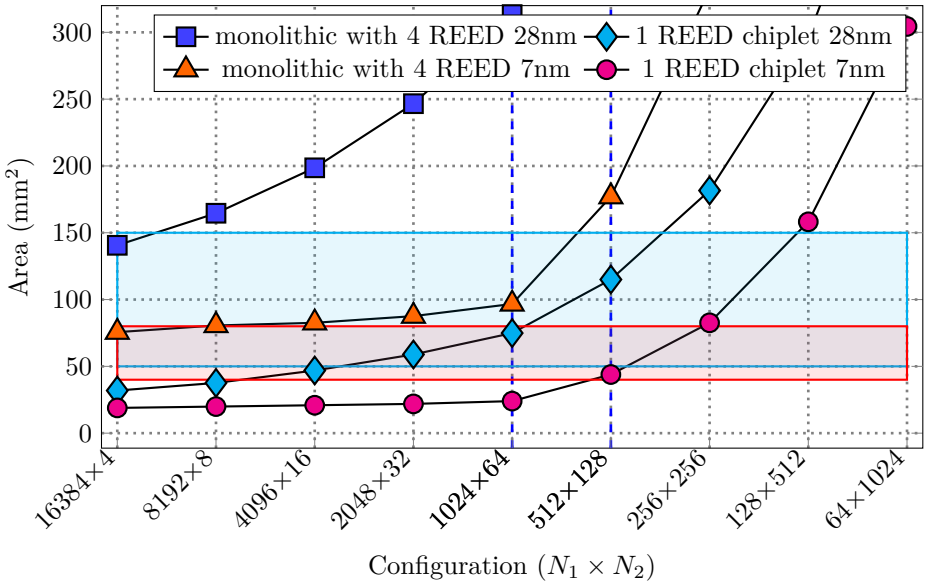

Figure 14: Area increase with rising throughput configurations [GPG23].

using identical instructions for software library and hardware implementation. However, in hardware, certain instructions can run simultaneously. Since they have constant execution time, we factored this behaviour into the software model, ensuring the clock cycle count aligned with expectations for hardware execution.

In Section 4.4, we elaborated on how our instruction-set architecture handles the micro (NTT, AUT, MAS) and macro (e.g. rotation, KeySwitch) instructions. Thus, a user does not need to handle micro-instructions due to the provided macro-instruction level abstraction. Predefined microcode for static macro-instructions ensures optimized data flow and memory management. Data exchange across chiplets occurs solely during KeySwitch and is incorporated into the microcode and task distribution. The simulator takes into account the bandwidth of C2C and HBM-chiplet communication along with data communication and distribution strategies (Section 5.3, Section 5.4). The run-time of macro-instructions is obtained using the known static schedule of micro-instructions. Finally, the macro-instructions are scheduled using OpenFHE [ABBB+22], and runtime is obtained for higher-level operations (bootstrapping, DNN).

We used the Cadence Genus tool to measure total power consumption for the entire REED-PU. For each operation, the duration for which any unit in the chiplet remains active is predetermined and static. The power consumption for each elementary building block is also obtained using the Cadence toolchain. We estimated the average power consumption based on this information combined with the runtime of each operation, following a methodology commonly employed by prior works such as [KLK+22, KKK+22, KKC+23, SFK+22] for estimating FHE hardware acceleration on ASIC technology. Similarly, in a multi-chiplet scenario, the operation runtime remains static, and we conservatively estimate communication overhead to be 2× slower than linear operations on the same volume of data. This approach provides a safe estimate of the total runtime. The cycle-accurate model serves as a simulator and effectively imitates the behaviour of the Instruction Set Controller within the chiplets, giving us a reasonable approximation of system performance. We validate this cycle-accurate modelling with Vivado simulation.

Table 5: The table compares memory (number of coefficients to store) and multiplier units for different N_1 and N_2 values with and without twiddle factor generation (TFG).

$N_1 \times N_2 \rightarrow$	2048×32	$\mathbf{1024 \times 64}$	512×128	256×256	128×512	64×1024
Total Mul.	432	**832**	1,600	3,072	5,888	11,264
TFG Mul.	68	**131**	258	513	1,024	2,047
TFG Mem.	222,912	**310,624**	486,400	707,232	1,149,248	1,771,488
~~TFG~~ Mul.	0	**0**	0	0	0	0
~~TFG~~ Mem.	4,260,320	**4,228,064**	4,212,704	4,206,560	4,206,560,	4,212,704
Mul. Inc.	16%	**16%**	16%	17%	17%	18%
Mem. Red.	95%	**93%**	88%	83%	73%	58%

6.2 What to expect from higher-throughput configurations?

Until now, we have examined two configurations (1024×64 and 512×128) that only partially demonstrate the advantages of our proposed scalable design methodology. As we double the throughput (by doubling the value of N_2), the area of PU only increases by approximately 1.5×. This trade-off arises because the chip area comprises two components— (i) the computation logic area, which scales linearly with throughput, and (ii) on-chip storage that remains fixed to a number of polynomials. When we opt for a higher configuration, the polynomial size remains the same while the number of coefficients to be processed in parallel increases. We also discuss storing versus generating twiddle factors on the fly. As shown in the Table 5, twiddle factor generation (TFG) increases the number of multipliers (Mul.) only up to 18% for different configurations (N_1 and N_2). Indeed, reduction in memory (Mem.) decreases (i.e., number of coefficients to store) with larger N_2. Yet, even for large values of N_2 (i.e., $N_1 = 64$ and $N_2 = 1024$), we observe a reduction in the memory by 57% compared to storing the twiddle factors (~~TFG~~).

However, an important question remains: *what configuration strikes the best balance between throughput and manufacturing cost?* To address this, we turn to [GPG23]. The authors report that the best manufacturing size for high yield ranges from 40 to 80 mm^2 for 7nm technology, while for 40nm, it ranges from 50 to 150 mm^2. In Figure 14, we present two sets of area consumption results for 28nm and 7nm technologies. The first set corresponds to four REED cores produced as a single monolithic chip, while the second set represents one REED chiplet. The best area ranges are highlighted in black and pink. As we can see, for both 7nm and 28nm, the configuration 512×128 falls within the best development area range and offers high throughput. The configuration 1024×64 is within the optimum range for 28nm and is close to it for 7nm. Monolithic designs, within the best range, offer 4× to 8× less throughput.

6.3 Comparison with Related Works

The realization of privacy-preserving computation through FHE holds great potential for the entire community, resulting in various acceleration works. Among these, the ASIC designs [FSK+21, KLK+22, KKK+22, SFK+22, KKC+23] have achieved the most promising acceleration results. However, a direct comparison with these works would be unfair as the benchmarks are provided for different parameters $(dnum, L, w, L_{boot})$. Hence, to ensure fairness, we also provide bootstrapping results for $dnum = 3$ (utilized by [KKC+23]) in Table 6. Next, since we cannot change the word size ($w = 54$) chosen for high precision, we select $L = 23, K = 8,$ and $L_{boot} = 17$ accordingly.

We use the amortized bootstrapping time $T_{A.S.}$ [KKK+22, KLK+22] metric that calculates the bootstrapping time divided by L_{eff} and packing n. This metric overlooks factors such as area, power, and precision. Higher precision necessitates a larger word size,

Table 6: Comparison of REED 2.5D with state-of-the-art

Work	**Area** (mm^2)	$\mathbf{T_{A.S.}}$ (ns)	$\mathbf{P_{Avg}}$ (W)	$\mathbf{EDAP}_w$ (/M)	**Parameters** ($N/L/dnum$)	**# B.S.**/Dollar (ops)* [GPG23]
$F1_{32}$	71.02†	470	28.5†	754.5	$2^{14}/23/24$	0.48
BTS_{64}	373.6	45.4	163.2	106.0	$2^{17}/39/2$	0.74
ARK_{64}	418.3	14.3	135	9.74	$2^{16}/23/4$	1.95
$CLake_{28}$	222.7†	17.6	124†	16.5	$2^{16}/60/1{-}3$	3.64
SH_{36}	178.8	12.8	94.7	4.1	$2^{16}/35/3$	6.23
SH_{64}	325.4	11.7	187	7.0	$2^{16}/22/3$	3.4
$\mathbf{REED}_{54}^{\ddagger(1)}$	**98.4****	**6.6**	**48.8**	**0.21**	$2^{16}/23/3$	**25.3**
	96.7	**28.8**	**49.4**	**3.96**	$2^{16}/30/31$	**6.32**
$\mathbf{REED}_{54}^{\ddagger(2)}$	**177**	**14.4**	**83.5**	**3.10**	$2^{16}/30/31$	**6.82**

† Area/power are normalized [NJO+17, KKC+23] (14nm/12nm to 7nm).
‡ Result for configuration (1) 1024×64 with one HBM per chiplet, and (2) 512×128 with two HBM per chiplet.
* The cost per mm^2 is obtained from [MUS] ($\$57,500/mm^2$ for 7nm) and yield estimates are taken from [GPG23, MWW+22] including manufacturing, pre- and post-testing, and integration costs.
** For $dnum < L + 1$, the area increases due to BaseConversion MAS units (Section A.1).

w (or expensive composite scaling). Thus, we use the EDAP (Energy-Delay-Area product) metric [LAS+09] and modify it ($EDAP_w$) to incorporate a linear increase due to word size (discussed in Section 7.1).

Table 6 compares our design's area consumption, performance, and power consumption for the packed bootstrapping operation with existing monolithic works, F1 [FSK+21], BTS [KKK+22], ARK [KLK+22], CraterLake (CLake) [SFK+22], and SHARP (SH) [KKC+23]. REED achieves 1.9× better performance than the state-of-the-art (SH_{36}) while consuming 1.8× less area. Our area consumption is less as the prior works utilize at least half of the chip area for on-chip memory. In our case, on-chip memory is not significant, as we utilize HBMs for major storage. We obtain better performance results due to the high throughput and 4.8× higher off-chip communication bandwidth offered by four HBM3 blocks – where each chiplet is exclusively connected to one HBM3.

Prior works utilize up to 1TB/s peak off-chip bandwidth, relying on large on-chip memories for storing keys, which necessitates much higher on-chip communication bandwidths—20TB/s [KLK+22], 36TB/s [KKC+23], and 84TB/s [SFK+22]. In contrast, although our proposed REED system uses 4.8× higher off-chip bandwidth (with each HBM offering 1.2TB/s at its peak), it efficiently offloads memory requirements to HBM, resulting in significantly reduced on-chip memory usage and consequently $4.2 - 7.5\times$ lower on-chip memory bandwidth requirements compared to [KLK+22, KKC+23, SFK+22]. The off-chip bandwidth requirement of one REED chiplet is comparable to that of a monolithic design. However, it incurs a 2.7× performance loss compared to the state-of-the-art [KKC+23] while consuming 3.36× less area. This trade-off is a consequence of designing chiplet-based architecture over a monolithic one. The REED 4-chiplet system offers better performance and 35× higher energy efficiency for lower chiplet area, as shown in Table 6.

We also assess the yield and manufacturing cost in Figure 15 by utilizing the results reported in [MWW+22, GPG23, MUS] including manufacturing, pre- and post-testing, and integration costs. For a fair comparison, we use the original area and not the word-size scaled area for prior works. As illustrated, we achieve the highest yield and lowest manufacturing cost on 7nm, resulting in the least overall cost (manufacturing cost/yield), 50% less than

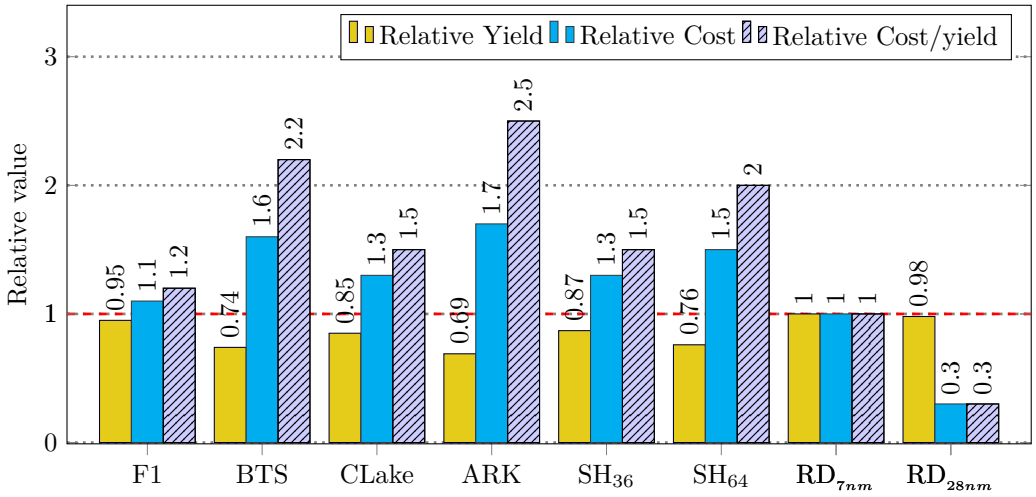

Figure 15: Relative a) yield of existing monolithic designs versus the proposed 7nm chiplet-based architecture [MWW+22], b) development cost (including Interposer cost) [GPG23, MUS, MAK+23], and c) cost of SiP development (cost/yield). RD refers to our work REED 2.5D.

state-of-the-art monolithic design SHARP_{64}. The cost is estimated using the yield metric provided in [MWW+22, FM22]. Based on this analysis, we observe that the yield for SHARP_{64} is approximately 73%, whereas the yield for the 4-chiplet REED (512×128) configuration reaches around 96%. This represents a 1.31× improvement in relative yield. Factoring in the manufacturing costs[MUS] and the chiplet packaging costs [GPG23], the design achieves an additional 1.5× cost reduction. Together, these factors contribute to a 50% reduction in the cost-to-yield ratio, offering an estimate of the overall chip manufacturing cost. On 28nm technology, we achieve 85% cheaper design compared to SHARP_{64}. Further, note that the cost per bootstrapping for $dnum = L + 1$ looks similar to prior monolithic works, but at this parameter, we also have the highest ($\approx 2\times$ more) computation depth remaining after bootstrapping for $\omega \geq 54$. Thus, applications utilizing this parameter choice require 2× less frequent bootstrapping operations.

6.4 Higher Chiplet-Integration Study

In Section 3, we examined multiple chiplet configurations and selected four ($\mathtt{r} = 4$) based on long-term utilization, lower power dissipation, and low integration costs. Our design methodology and data/task distribution approach remain adaptable to any desired chiplet configuration and the number of chiplets. In fully connected chiplet nodes, interconnect length between chiplets can significantly impact energy consumption and latency. Therefore, our proposed non-blocking ring-based communication technique for KeySwitching shows a better advantage for higher chiplet integration density.

Table 7: Results for 1024×64 configuration on 28nm technology with parameters ($N/L/dnum = 2^{16}/30/31$).

Number of Chiplets →	**4**	**8**	**12**
Area (mm^2)	311.9	623.8	935.7 (> reticle limit)
$\mathbf{T_{A.S.}}$ (ns)	14.41	7.37	4.98

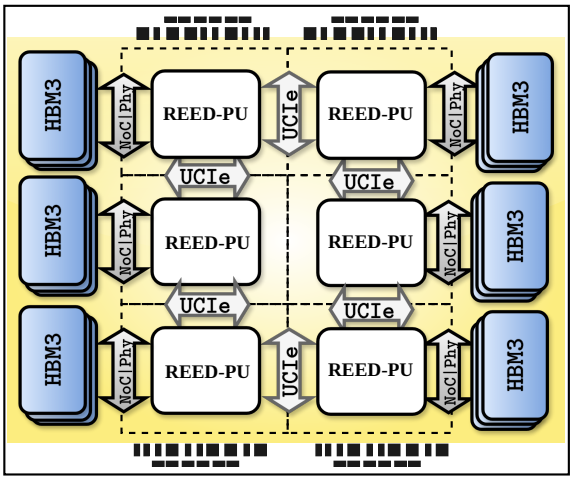

Figure 16: The complete architecture diagram of 6-chiplet REED 2.5D for 1024×64 configuration.

Figure 16 illustrates a chiplet-based architecture for six chiplets, which can be expanded to accommodate more chiplets. Additionally, energy-efficient lower-bandwidth memories, such as DDR, can be integrated with the appropriate (N_1, N_2) configuration based on memory throughput. While increasing the number of chiplets enhances performance, as shown in Table 7, it comes at the cost of area and underutilization when the current multiplicative depth (l) falls below the number of chiplets ($l < r$). However, more chiplets can lead to better performance in the long term and allow tiling beyond the reticle limit, albeit with the additional area, power, and integration overhead. Also, note that a higher number of chiplet interconnects implies additional points of failure, making testability and reliability more involved. A more detailed multi-chiplet study is also presented in concurrent work [KKCA24], where the authors show that increasing the cores from 4 to 16 results in speedup of only $1.99 - 2.11\times$.

7 Application benchmarks

We benchmark three machine learning applications: linear regression, logistic regression, and a Deep Neural Network (DNN). Each application is evaluated for *encrypted* training and inference. In this setting, the server provides computational support without knowledge of the data or model parameters, ensuring complete blind computation. Most applications benchmarked in the previous works [FSK+21] are partially blind; the server does not see the data but knows the model parameters to evaluate it. To the best of our knowledge, none of the previous works benchmark an encrypted neural network training. The speedup results are presented in Table 8 using the most area conservative design (1024×64). Higher configuration (512×128) will improve the performance by 2×.

- **Linear Regression**: We employ the Kaggle Insurance dataset [Sco20] to benchmark linear regression. The model uses a batch size of 1204 and 1338 input feature vectors (each containing six features) for training and inference and achieves an accuracy of 78.1% (same as plain model [Sco20]).
- **Logistic Regression**: It is a supervised machine learning model that utilizes the log function, evaluated using function approximations in a homomorphic context. Its accuracy depends on the degree of approximation function expansion and precision. Existing works, such as [SFK+22, KKC+23], utilize the HELR [HHCP19] to

Table 8: Application benchmark and the speedup achieved by REED 2.5D. The CPU speed is reported on a 24-core, 2×Intel Xeon CPU X5690 @ 3.47GHz with 192GB DDR3 RAM.

Appl.	Accuracy	Op	Time		Speedup
			CPU	HW	
Lin.Reg.	78.12%	Inf.	0.86 s	0.31 ms	2,873×
		Trn.	13.82 s	4.6 ms	2,991×
Log.Reg.	61.8%	Inf.	1.27 s	0.46 ms	2,785×
		Trn.	11.18 s	3.8 ms	2,865×
DNN	95.2%	Inf.	128.7 s	48.6 ms	2,646×
		Trn.	29 days	920 s	2,725×

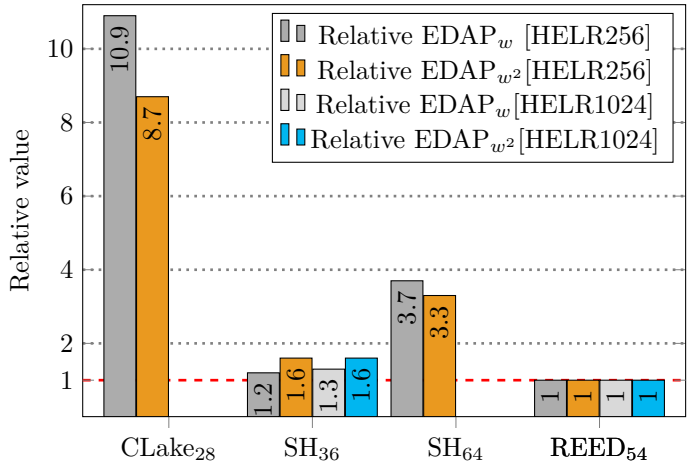

Figure 17: Relative metrics comparison for the HELR [HHCP19] application with batch sizes 256 and 1024. Under these metrics, the lower the value, the better.

benchmark encrypted training on MNIST [LC10] data, with batch sizes (256, 1024). In Figure 17, we illustrate the performance advantage of REED 2.5D. To predict cancer probability, we further evaluate logistic regression on the iDASH2017 cancer dataset (similar to [KSK+18]). Here, we achieve a training accuracy of 62% in a single iteration. This dataset comprises 18 features per input, with batch sizes of 1422 and 1579 used for training and inference.

- **Deep Neural Network** : The DNN serves as a powerful tool for Deep Learning. Our study employs a DNN, shown in Figure 18, for the MNIST dataset [LC10], with two hidden and one output layer. We pack four pre-processed images per batch to prevent overflow during matrix multiplication. DNN training requires 12,500 batches. Thus, all the existing works [KLK+22, KKK+22, SFK+22, KKC+23] not providing computation-communication parallelism will suffer as their on-chip memory is insufficient. The DNN is trained for ≈7000 (≈5.8 Bootstrappings per iteration) iterations and achieves 95.2% accuracy in 29 days using OpenFHE [ABBB+22]. REED 2.5D could finish this in only 15.4 minutes. This is where our computation-communication parallelism shines, as many ciphertexts are required for such an application. None of the works in literature offers this and is bound to suffer for memory-intensive applications.

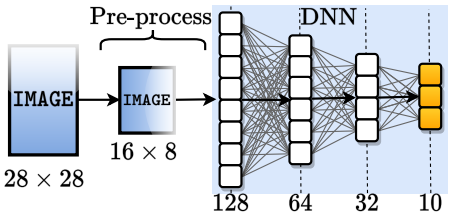

Figure 18: A DNN for MNIST [LC10] with two hidden and one output layers.

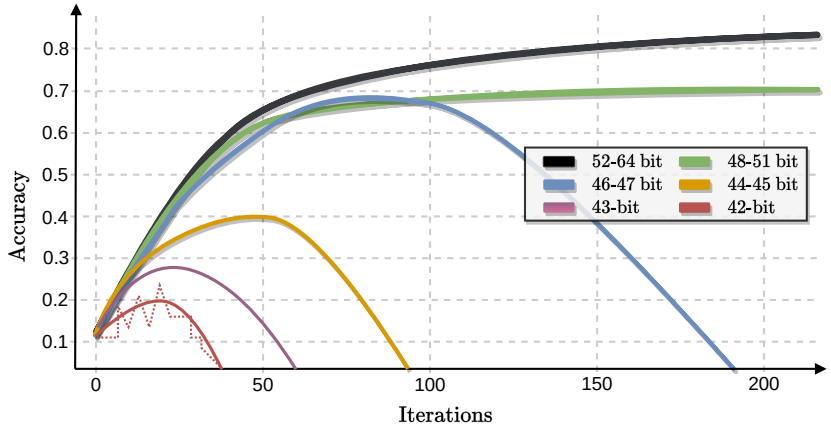

Figure 19: Accuracy plot of different word sizes for the DNN. The lines are smoothened and the red dotted zig-zag line resembles the original form.

7.1 Precision-loss Experimental Study

Another facet of privacy-preserving computation is precision loss. Since the server cannot see the intermediate or final results, the best it can do is to ensure that the parameters it operates on support higher precision. To validate our parameter sets, we ran experiments for the DNN training. In Figure 19, we can see how quickly the training accuracy drops as the word size is reduced. Thus, precision plays a vital role in providing privacy-preserving computation on the cloud. Our choice of 54-bit word size strikes the perfect balance between precision and performance. Works offering a smaller word-size [FSK+21, SFK+22, KKC+23] require in-depth study to mitigate the accuracy loss due to low precision.

8 Future Scope: Journey from 2.5D to 3D

The extension of REED 2.5D to a complete 3D IC holds immense potential for future computing. To achieve this transition, we have two options: connecting the PU with the HBM controller via TSV (as shown in Figure 20) or merging the PU unit with the lower HBM controller die. Since HBM is sold as an IP, the latter approach relies on the IP vendors to integrate the PU. By adopting either of these approaches, we can significantly reduce the reliance on the Network-on-Chip (NoC), leading to a compact chip design with lower power consumption. Each chiplet will be a full 3D IC package (PU and Memory) and will need a C2C link via interposer for connecting to other chiplets. A reduction in the area is expected due to fewer HBM stacks on the lateral area and the integration of the REED-PU unit with the HBM controller. Additionally, decreased critical paths would further enhance the design's performance. Thus, the REED's 3D IC integration promises a huge reduction in overall chip area and power consumption.

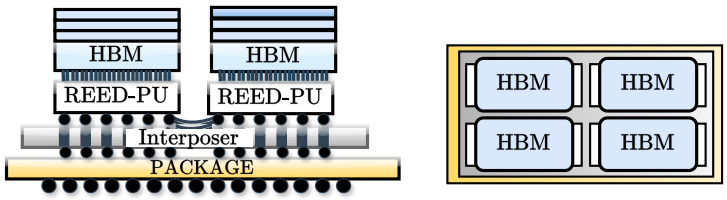

Figure 20: The side and top view of futuristic RE3D has four REED 3DIC chiplets.

9 Conclusion

FHE has garnered considerable interest due to its ability to preserve data and computation privacy, leading to several efforts for accelerating FHE using large monolithic ASIC designs. However, many of these attempts primarily focus on acceleration at the expense of yield, manufacturing cost, and scalability. We proposed a scalable FHE accelerator design methodology for a multi-chiplet system that can be easily extended to larger configurations while adapting to constrained environments.

Chiplet-based designs face inherent challenges, such as increased latency costs due to slow C2C communication. We designed REED to address these and show advantages over the monolithic designs in terms of performance, area, and energy consumption. REED achieved this feat by utilizing a non-trivial yet uncomplicated bandwidth-oriented design methodology and a modular design approach.

An efficient workload division strategy optimized for multi-chiplet architectures is proposed to minimize memory usage and enhance efficiency through interleaved data and workload distribution strategy for all FHE routines. Next, to address slow chiplet-to-chiplet communication, a novel non-blocking ring-based inter-chiplet communication strategy tailored to FHE is introduced. Additionally, a scalable bandwidth-oriented design methodology is adopted, offering flexibility to adjust to the varying area and performance needs while supporting communication-computation parallelism within each chiplet. Furthermore, novel design techniques are presented for building blocks (NTT/AS/AUT), enhancing scalability. These techniques accelerate routines such as KeySwitch or Bootstrapping and reduce their latency by ≈67%. Finally, REED benchmarks an encrypted DNN training, demonstrating utility for real-world FHE applications.

All the REED-chiplets are small and identical, allowing us to prototype the building blocks using FPGA and validate the functionality, which had not been done by prior works. Summarily, the proposed design methodology for REED showcased a robust and scalable approach to FHE acceleration that addressed several key challenges inherent in traditional monolithic designs. This paves the way for interesting future prospects such as formal verification. The advancements presented in this work hold the promise of advancing privacy-preserving computations and promoting the wider adoption of fully homomorphic encryption.

Acknowledgement

This work was supported in part by Samsung Electronics Co. Ltd., Samsung Advanced Institute of Technology and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark. We extend our gratitude to the anonymous reviewers for their constructive feedback. We thank Ian Khodachenko for conducting extensive application benchmarking, the results of which were integral to this paper, and Florian Hirner for FPGA prototyping.

Appendix

A KeySwitch Task Distribution for $dnum < L + 1$ ($k > 1$)

Algorithm 10 `CKKS.KeySwitch` [HK19, KPP22] (for arbitrary $dnum$)

In: $\mathtt{d} = (\tilde{\mathbf{d}}_0, \tilde{\mathbf{d}}_1, \tilde{\mathbf{d}}_2) \in R^3_{Q_l}$, $\tilde{\mathtt{ksk}}_0 \in R^{dnum}_{PQ_l}, \tilde{\mathtt{ksk}}_1 \in R^{dnum}_{PQ_l}$
Out: $\mathtt{d}' = (\tilde{\mathbf{d}}'_0, \tilde{\mathbf{d}}'_1) \in R^2_{Q_l}$

for $j = 0$ to $dnum - 1$ **do**
  $\tilde{\mathbf{y}}[j] \leftarrow \tilde{d}_2[j \cdot K : (j+1) \cdot K - 1] \cup \mathtt{BConvRout}_{Q_K \rightarrow PQ_{l/K}}(\tilde{d}_2[j \cdot K : (j+1) \cdot K - 1])$
  $(\tilde{\mathbf{c}}''_0[j], \tilde{\mathbf{c}}''_1[j]) \leftarrow 0$
  for $i = 0$ to $l + K$ **do**
    $\tilde{c}''_0[j][i] \leftarrow \left[\tilde{c}''_0[j][i] + \tilde{\mathtt{ksk}}_0[j][i] \cdot \tilde{y}[j][i]\right]_{q_j}$
    $\tilde{c}''_1[j][i] \leftarrow \left[\tilde{c}''_1[j][i] + \tilde{\mathtt{ksk}}_1[j][i] \cdot \tilde{y}[j][i]\right]_{q_j}$
  end for
end for
for $j = 1$ to $dnum - 1$ **do**
  $\tilde{\mathbf{c}}''_0[0] \leftarrow \left[\tilde{\mathbf{c}}''_0[0] + \tilde{\mathbf{c}}''_0[j]\right]$
  $\tilde{\mathbf{c}}''_1[0] \leftarrow \left[\tilde{\mathbf{c}}''_1[0] + \tilde{\mathbf{c}}''_0[j]\right]$
end for
$\tilde{\mathbf{d}}'_0 \leftarrow \tilde{\mathbf{d}}_0 + (\tilde{\mathbf{c}}''_0[0]_{Q_l} - \mathtt{BConvRout}_{P \rightarrow Q_l}(\tilde{\mathbf{c}}''_0[0]))$
$\tilde{\mathbf{d}}'_1 \leftarrow \tilde{\mathbf{d}}_1 + (\tilde{\mathbf{c}}''_1[0]_{Q_l} - \mathtt{BConvRout}_{P \rightarrow Q_l}(\tilde{\mathbf{c}}''_1[0]))$

The generic version of the KeySwitch routine [HK19, KPP22] is presented in Algorithm 10. The $L + 1$ limbs of the ciphertext are split into $dnum$ digits of K limbs. Every digit consisting of K limbs is used to obtain $l + K + 1$ limbs after the ModUp operation (via BConvRout Algorithm 11) for ciphertext at depth l. This results in $dnum \cdot (l + K + 1)$ limbs, which are then multiplied with the keys and accumulated to return $(l + K + 1)$ limbs per ciphertext component. Finally, a ModDown is done to reduce the limbs to $(l + 1)$ limbs and the operation flow is very similar to that of ModUp. As discussed in Section 5.2, maintaining the limb-based decomposition proves advantageous compared to switching between limb-based and coefficient-based methods. We now explore how this limb-based decomposition applies when $dnum < L + 1$. It is important to note that for $dnum < L + 1$, NTT computation cannot start until the INTT results are multiplied with base hats $(\hat{p}_i, {\hat{p}_i}^{-1})$ for BaseConversion.

The first technique involves digit-wise distributing data and tasks across limbs, where the number of chiplets equals $dnum$. This allows each chiplet to independently execute the outer loop for ModUp in Algorithm 10 without requiring inter-chiplet data exchange. Data is only shared after ModUp (before Key Multiplication) and during ModDown, as illustrated in Figure 21 first figure. The authors in [KLK+22], note that this has an overhead of $2 \cdot dnum \cdot (K + l + 1)$ polynomials per chiplet, which can be reduced to $\frac{2 \cdot (dnum-1) \cdot (l+K+1)}{dnum}$ for ModUp as explained in Section 5.2. During ModDown, $2 \cdot K$ polynomials would be communicated for BaseConversion. ModDown can also be handled within each chiplet, eliminating the need for cross-chiplet data exchange if the result of Key Multiplication is duplicated across all chiplets, as depicted in the second figure of Figure 21. This results in a one-time communication cost of $\frac{2 \cdot (dnum-1) \cdot (l+1)}{dnum} + 2 \cdot K$ polynomials per chiplet and requires additional storage for $2 \cdot (K - 1)$ polynomials in each chiplet.

Each chiplet would have to perform duplicate K INTT operations and BaseConversion steps during ModDown while the $(l + 1)$ NTT operations would be distributed across the chiplets. A key limitation of this technique is that not much computation can be done in parallel with the communication after ModUp. While the polynomial transfers

Algorithm 11 $\texttt{CKKS.BconvRout}_{P \to Q_l}$

In: $\tilde{\mathbf{y}}_2 \in R_P$
Out: $\tilde{\mathbf{y}} \in R_{Q_l}$

1: **for** $j = 0$ to $K-1$ **do**
2: $\quad y_2[j]_{p_j} \leftarrow \texttt{INTT}(\tilde{y}_2[j])$
3: **end for**
4: **for** $j = 0$ to l **do**
5: $\quad y[j]_{q_j} \leftarrow \sum_{i=0}^{K-1}(y_2[i] \cdot \hat{p}_i^{-1} \bmod p_i) \cdot \hat{p}_i \bmod q_j$ ▷ BaseConversion
6: $\quad \tilde{y}[j]_{q_j} \leftarrow \texttt{NTT}(y[j+K])_{q_j}$
7: **end for**

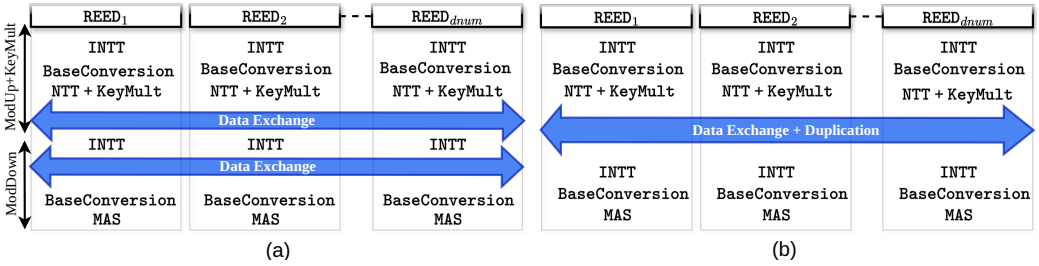

Figure 21: The Data and Task distribution when limbs are distributed digit-wise for KeySwitch computation. The INTT, NTT, and BaseConversion mentioned here refer to computations required per digit. In the first case, Data Exchange is done in parallel with BaseConversion, and in the second case, it is done in parallel with KeyMult. This does not produce a full decoupling effect when C2C communication is slower.

can overlap with the NTT and Key Multiplication computations, the high throughput of NTT+KeyMult operations means this overlap does not achieve the desired decoupling effect. As a result, the potential for parallelism is limited, leading to delays. Additionally, this approach demands many chiplets for higher values of *dnum*. As the digits for the key switch decrease, many chiplets become idle, further reducing efficiency. This inefficiency in fully utilizing chiplets highlights a bottleneck in achieving optimal performance.

In contrast, the alternate limb distribution technique distributes the task for each digit across a fixed number of chiplets, as illustrated in Figure 22. This means communication is only required during ModUp/ModDown, after which all the data needed for Key Multiplication and accumulation remains confined within a single chiplet. During ModUp, the chiplets broadcast the INTT results (similar to the case of $dnum = L+1$), which is efficient as only $l+1$ polynomials undergo INTT conversion. After ModUp, the number of polynomials increases to $dnum \cdot (l+K+1)$. Although communication is necessary for both ModUp and ModDown, our technique minimizes communication overhead since the number of polynomials broadcasted during ModUp $(l+1)$ and ModDown $(2 \cdot K)$ is significantly lower compared to the previous method's $\frac{2 \cdot (dnum-1) \cdot (l+1)}{dnum} + 2 \cdot K$ for $dnum > 2$.

We investigated the possibility of using data duplication to decouple INTT data transfer from NTT computation. However, we found that the overhead caused by duplicating data after each operation outweighed any potential communication benefits during ModUp and ModDown. Thus, we converge to an approach where the chiplets do not compute only K INTT required for each digit computation during ModUp but all $l+1$ INTT operations, which they would require for computation corresponding to all the digits.

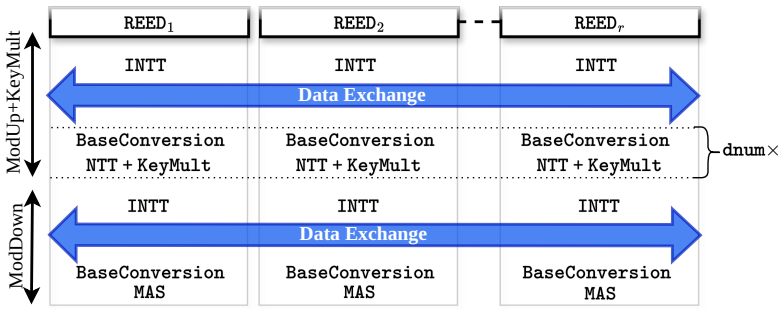

Figure 22: The Data and Task distribution when limbs are distributed alternately for KeySwitch computation. The Data Exchange is done in parallel with INTT, BaseConversion, and NTT+KeyMult of previous digits.

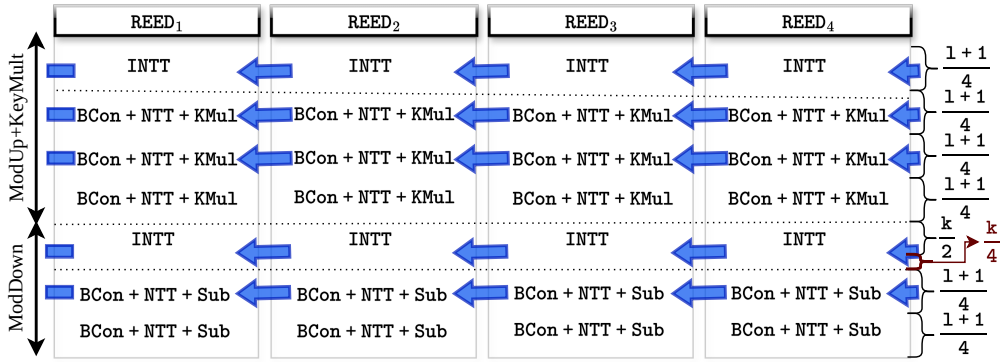

Figure 23: KeySwitch Flow showing computation throughput and communication overhead for ModUp, KeyMult and ModDown for $dnum = 3$.

A.1 Throughput Computation of KeySwitch for $dnum < L + 1$

In this section, we discuss how the throughput is obtained when $dnum = 3$, in line with Section 4.6, in the four ($= r$) chiplet setting of REED. As discussed above we, perform all $l + 1$ INTT at once, to bridge the computation communication gap. Thus, each chiplet performs at most $\frac{l+1}{r}$ INTT. For the first set of BaseConversion followed by Key Multiplication, only $\frac{l+1}{dnum}$ INTT results are needed. Since, these limbs are distributed across chiplets a communication of $\frac{(r-1)\cdot(l+1)}{dnum\cdot r}$ is required to every chiplet. This communication also works in ring manner, similar to the strategy proposed in Section 5.4.

Note that we cannot start NTT before we have accumulated all the $\frac{l+1}{dnum}$ results as required for Algorithm 11. When $dnum = L + 1$, this value is 1, hence we can start the NTT immediately on the INTT result, however, the same cannot be done for lower values of $dnum$. Thus, in the prior case, while a long communication window was available, here the communication window is limited. The first set of $\frac{(r-1)\cdot(l+1)}{dnum\cdot r}$ INTT results have to be communicated while the chiplet is computing $\frac{l+1}{r}$ INTTs. Each BaseConversion computes $l + 1$ NTTs ($\frac{l+1}{r}$ NTTs), which gives $\frac{dnum}{r-1}\times$ larger communication window for the INTT results required for subsequent BaseConversions. Note that, for $dnum = 3$ and $r = 4$, its value is one, hence we only face computation overhead for the last polynomial as the C2C communication offers the same throughput as the configuration 1024×64. However, when the C2C communication becomes slower, than this directly impacts the communication and hence the delay before the BaseConversions.

Now, the goal of computation is to prevent computation from becoming the bottleneck in this case. Thus, instead of sending plain INTT results, the results are multiplied with $\hat{\rho}_i^{-1}$ for Step 5 [KLK+22], in Algorithm 11. After this the chiplets only need to perform a MAS operation on all the INTT results for each new base during BaseConversion. We are only restricted by off-chip communication bandwidth, and therefore we wait until all the polynomials have been shared. The polynomials in INTT form are in on-chip memory. The memory for each polynomial is split across various SRAMs. Therefore, one polynomial can be loaded at a higher throughput. The operations on these can be handled by an extra MAS unit for BaseConversion using $K\times$ ($K = 8$ for $dnum = 3$) more multipliers to offer the same throughput as NTT. Thus, the BaseConversion, followed by NTT, and Key Multiplication and accumulation works in a pipeline. The key multiplication with the pre-existing K bases is done in parallel to their INTT conversion, as it does not require any pre-computation. Hence, the overhead of the ModUp+KeyMul is directly correlated to the computation time of the INTT polynomials $\frac{(l+1)}{r}$ and the $dnum$ BaseConversions $\frac{dnum\cdot(l+1)}{r}$, as shown in Figure 23.

While, the communication could be completely decoupled for the ModUp+KeyMul step, the decoupling reduces for ModDown. Each chiplet computes the $\frac{K}{r}$ INTT results and requires communication of $\frac{(r-1)\cdot K}{r}$ INTT results before it can perform the BaseConversion requiring $\frac{(l+1)}{r}$ NTT computations per chiplet and the subtraction in pipeline. Thus, the overall overhead of this step is determined by communication of $\frac{(r-1)\cdot K}{r}$ polynomials and followed by $\frac{(l+1)}{r}$ NTT computation. This is incurred twice, once for each ciphertext component. Instead of doing the ModDown sequentially for each ciphertext component, we follow the technique proposed earlier and perform all the INTT $\frac{2\cdot K}{r}$ INTT computations at once. Thus the communication overhead for the first ModDown is reduced to $\frac{(r-3)\cdot K}{r}$. While the first ModDown is being computed the limbs for the next ModDown are communicated. Thus, the delay due to communication overhead of $\frac{(r-3)\cdot K}{r}$ is incurred only once, and the computation overhead is $\frac{2\cdot(l+1+K)}{r}$ NTT/INTT computations.

Overall, this results in runtime complexity of $\frac{(2\cdot(l+1+K)+(dnum+1)\cdot(l+1)+(r-3)\cdot K)}{r}$. With $dnum = 3$ and $r = 4$, the communication overhead is ≈5%, while INTT/NTT is being computed in parallel and MAS operations in pipeline. Substituting the values of $dnum = L+1$, shows how this reduces to bare minimum. Thus, NTT unit stays almost fully utilized (≈95%) during the hybrid key-switch for $dnum = 3$. We have kept our analysis generic, so that similar results can be derived for varying values of $dnum$ and r. This also shows how $r = 4$, gives a sweet spot, and a higher value would result in higher communication delay, not only during ModDown but also during ModUp, for lower values of $dnum$. Note that, this 5% communication delay can be completely bridged by interleaving the last BaseConversion+NTT+KeyMul step with the INTT required for ModDown, as no communication happens during this step. However, we leave this dataflow utilization for future works.

References

[ABBB+22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptog-*

raphy, WAHC'22, pages 53–63, New York, NY, USA, 2022. Association for Computing Machinery.

[AMD23] AMD. Amd instinct™ mi300 series accelerators. Technical report, AMD, 2023.

[BDTV23] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 741–755. ACM, 2023.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electron. Colloquium Comput. Complex.*, page 111, 2011.

[BHM+20] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.

[BMTH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 587–617. Springer, 2021.

[BZP+23] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. HE3DB: an efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2930–2944. ACM, 2023.

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved Bootstrapping for Approximate Homomorphic Encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[CHK+18a] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for Approximate Homomorphic Encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2018.

[CHK+18b] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368. Springer, 2018.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CLSW11] Kun-Chih Chen, Shu-Yen Lin, Wen-Chung Shen, and An-Yeu Wu. A scalable built-in self-recovery (BISR) VLSI architecture and design methodology for 2d-mesh based on-chip networks. *Des. Autom. Embed. Syst.*, 15(2):111–132, 2011.

[CPS+23] Kwanyeob Chae, Jiyeon Park, Jaegeun Song, Billy Koo, Jihun Oh, Shinyoung Yi, Won Lee, Dongha Kim, Taekyung Yeo, Kyeongkeun Kang, Sangsoo Park, Eunsu Kim, Sukhyun Jung, Sanghune Park, Sungcheol Park, Mijung Noh, Hyo-Gyuem Rhew, and Jongshin Shin. A 4nm 1.15TB/s HBM3 Interface with Resistor-Tuned Offset-Calibration and In-Situ Margin-Detection. In *IEEE International Solid- State Circuits Conference, ISSCC 2023, San Francisco, CA, USA, February 19-23, 2023*, pages 406–407. IEEE, 2023.

[CVS+16] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[EH22] Anne C. Elster and Tor A. Haugdahl. Nvidia Hopper GPU and Grace CPU Highlights. *Computing in Science & Engineering*, 24(2):95–100, 2022.

[FM22] Yinxiao Feng and Kaisheng Ma. Chiplet actuary: a quantitative cost model and multi-chiplet architecture exploration. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*, pages 121–126. ACM, 2022.

[FSK+21] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version), 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[FWX+23] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. Tensorfhe: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 922–934. IEEE, 2023.

[Gar59] Harvey L. Garner. The Residue Number System. *IRE Trans. Electron. Comput.*, 8(2):140–147, 1959.

[GBP+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, USA, 2009.

[Gon21] Joe L. Gonzalez. *Heterogeneous Integration of Chiplets Using Socketed Platforms, Off-Chip Flexible Interconnects, and Self-Alignment Technologies*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2021.

[GPG23] Alexander Graening, Saptadeep Pal, and Puneet Gupta. Chiplets: How Small is too Small? *ACM/IEEE Design Automation Conference (DAC),*, 2023.

[HHCP19] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 9466–9471. AAAI Press, 2019.

[HK19] Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. Cryptology ePrint Archive, Report 2019/688, 2019. https://ia.cr/2019/688.

[HKKR20] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*, pages 968–981. IEEE, 2020.

[IBM20] IBM. IBM Cost of a Data Breach 2022 – Highlights for Cloud Security Professionals. *Technical Report*, 2020.

[Int23] Intel. Intel xeon cpu max 9400. Technical report, Intel, 2023.

[JED22] JEDEC. High Bandwidth Memory DRAM (HBM3). *Tech. Rep. JESD238*, 2022.

[JKA+21] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):114–148, Aug. 2021.

[JKLS18] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1209–1222. ACM, 2018.

[JLK+21] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access*, 9:98772–98789, 2021.

[KKC+23] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In Yan Solihin and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 18:1–18:15. ACM, 2023.

[KKCA24] Sangpyo Kim, Jongmin Kim, Jaeyoung Choi, and Jung Ho Ahn. Cifher: A chiplet-based FHE accelerator with a resizable structure. In *International Symposium on Secure and Private Execution Environment Design, SEED 2024, Orlando, FL, USA, May 16-17, 2024*, pages 119–130. IEEE, 2024.

[KKK+22] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 711–725, New York, NY, USA, 2022. Association for Computing Machinery.

[KKL+23a] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 726–740. ACM, 2023.

[KKL+23b] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 726–740. ACM, 2023.

[KLK+22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse, 2022.

[KMP+21] Gokul Krishnan, Sumit K. Mandal, Manvitha Pannala, Chaitali Chakrabarti, Jae-Sun Seo, Ümit Y. Ogras, and Yu Cao. SIAM: chiplet-based scalable in-memory acceleration with mesh for deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 20(5s):68:1–68:24, 2021.

[KPP22] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. Approximate homomorphic encryption with reduced approximation error. In Steven D. Galbraith, editor, *Topics in Cryptology - CT-RSA 2022 - Cryptographers' Track at the RSA Conference 2022, Virtual Event, March 1-2, 2022, Proceedings*, volume 13161 of *Lecture Notes in Computer Science*, pages 120–144. Springer, 2022.

[KSK+18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11, 10 2018.

[LAS+09] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In David H.

Albonesi, Margaret Martonosi, David I. August, and José F. Martínez, editors, *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 469–480. ACM, 2009.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[LLL+15] Dong-Uk Lee, Kang Seol Lee, Yongwoo Lee, Kyung Whan Kim, Jong-Ho Kang, Jaejin Lee, and Jun Hyun Chun. Design considerations of HBM stacked DRAM and the memory architecture extension. In *2015 IEEE Custom Integrated Circuits Conference, CICC 2015, San Jose, CA, USA, September 28-30, 2015*, pages 1–8. IEEE, 2015.

[LSL20] James Andrew Lewis, Zhanna L. Malekos Smith, and Eugenia Lostri. The Hidden Costs of Cybercrime. *Technical Report*, 2020.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuzk: Accelerating zero-knowledge proof with A faster parallel multi-scalar multiplication algorithm on gpus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):194–220, 2023.

[MAK+23] Ahmet Can Mert, Aikata, Sunmin Kwon, Youngsam Shin, Donghoon Yoo, Yongwoo Lee, and Sujoy Sinha Roy. Medha: Microcoded Hardware Accelerator for computing on Encrypted data. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(1):463–500, 2023.

[Man22] Tobias Mann. Amd was right about chiplets, intel's gelsinger all but says. Technical report, AMD, 2022.

[MNLK23] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. Level up: Private non-interactive decision tree evaluation using levelled homomorphic encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2945–2958. ACM, 2023.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[Mor20] Steve Morgan. McAfee Vastly Underestimates The Cost Of Cybercrime. *Cybersecurity Report*, 2020.

[MÖS19] Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. Design and implementation of a fast and scalable ntt-based polynomial multiplier architecture. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 253–260. IEEE, 2019.

[Mur23] Brett Murdock. What Designers Need to Know About HBM3. *Synopsys*, Accessed on July 11, 2023.

[MUS] MUSE Semiconductor. TSMC UNIVERSITY FINFET PROGRAM. https://www.musesemi.com/university-finfet-program. Accessed July 27th 2023.

[MWW+22] Xiaohan Ma, Ying Wang, Yujie Wang, Xuyi Cai, and Yinhe Han. Survey on chiplets: interface, interconnect and integration methodology. *CCF Trans. High Perform. Comput.*, 4(1):43–52, 2022.

[NJO+17] S. Narasimha, B. Jagannathan, A. Ogino, D. Jaeger, B. Greene, C. Sheraw, K. Zhao, B. Haran, U. Kwon, A. K. M. Mahalingam, B. Kannan, B. Morganfeld, J. Dechene, C. Radens, A. Tessier, A. Hassan, H. Narisetty, I. Ahsan, M. Aminpur, C. An, M. Aquilino, A. Arya, R. Augur, N. Baliga, R. Bhelkar, G. Biery, A. Blauberg, N. Borjemscaia, A. Bryant, L. Cao, V. Chauhan, M. Chen, L. Cheng, J. Choo, C. Christiansen, T. Chu, B. Cohen, R. Coleman, D. Conklin, S. Crown, A. da Silva, D. Dechene, G. Derderian, S. Deshpande, G. Dilliway, K. Donegan, M. Eller, Y. Fan, Q. Fang, A. Gassaria, R. Gauthier, S. Ghosh, G. Gifford, T. Gordon, M. Gribelyuk, G. Han, J.H. Han, K. Han, M. Hasan, J. Higman, J. Holt, L. Hu, L. Huang, C. Huang, T. Hung, Y. Jin, J. Johnson, S. Johnson, V. Joshi, M. Joshi, P. Justison, S. Kalaga, T. Kim, W. Kim, R. Krishnan, B. Krishnan, K. Anil, M. Kumar, J. Lee, R. Lee, J. Lemon, S.L. Liew, P. Lindo, M. Lingalugari, M. Lipinski, P. Liu, J. Liu, S. Lucarini, W. Ma, E. Maciejewski, S. Madisetti, A. Malinowski, J. Mehta, C. Meng, S. Mitra, C. Montgomery, H. Nayfeh, T. Nigam, G. Northrop, K. Onishi, C. Ordonio, M. Ozbek, R. Pal, S. Parihar, O. Patterson, E. Ramanathan, I. Ramirez, R. Ranjan, J. Sarad, V. Sardesai, S. Saudari, C. Schiller, B. Senapati, C. Serrau, N. Shah, T. Shen, H. Sheng, J. Shepard, Y. Shi, M.C. Silvestre, D. Singh, Z. Song, J. Sporre, P. Srinivasan, Z. Sun, A. Sutton, R. Sweeney, K. Tabakman, M. Tan, X. Wang, E. Woodard, G. Xu, D. Xu, T. Xuan, Y. Yan, J. Yang, K.B. Yeap, M. Yu, A. Zainuddin, J. Zeng, K. Zhang, M. Zhao, Y. Zhong, R. Carter, C.-H. Lin, S. Grunow, C. Child, M. Lagus, R. Fox, E. Kaste, G. Gomba, S. Samavedam, P. Agnello, and D. K. Sohn. A 7nm cmos technology platform for mobile and high performance compute application. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 29.5.1–29.5.4, 2017.

[NSA+22] Mohammed Nabeel, Deepraj Soni, Mohammed Ashraf, Mizan Abraha Gebremichael, Homer Gamil, Eduardo Chielle, Ramesh Karri, Mihai Sanduleanu, and Michail Maniatakos. CoFHEE: A Co-processor for Fully Homomorphic Encryption Execution, 2022.

[PLC+23] Myeong-Jae Park, Jinhyung Lee, Kyungjun Cho, Ji Hwan Park, Junil Moon, Sung-Hak Lee, Tae-Kyun Kim, Sanghoon Oh, Seokwoo Choi, Yongsuk Choi, Ho Sung Cho, Tae-Sik Yun, Young Jun Koo, Jae-Seung Lee, Byung Kuk Yoon, Young Jun Park, Sangmuk Oh, Chang Kwon Lee, Seong-Hee Lee, Hyun-Woo Kim, Yucheon Ju, Seung-Kyun Lim, Kyo Yun Lee, Sang-Hoon Lee, Woo Sung We, Seungchan Kim, Seung Min Yang, Keonho Lee, In-Keun Kim, Younghyun Jeon, Jae-Hyung Park, Jong Chan Yun, Seonyeol Kim, Dong-Yeol Lee, Su-Hyun Oh, Junghyun Shin, Yeonho Lee, Jieun Jang, and Joohwan Cho. A 192-Gb 12-High 896-GB/s HBM3 DRAM With a TSV Auto-Calibration Scheme and Machine-Learning-Based Layout Optimization. *IEEE J. Solid State Circuits*, 58(1):256–269, 2023.

[PMSW18] Nicolas Papernot, Patrick D. McDaniel, Arunesh Sinha, and Michael P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 399–414. IEEE, 2018.

[PZM+23] Gianna Paulin, Florian Zaruba, Stefan Mach, Manuel Eggimann, Matheus Cavalcante, Paula Scheffler, Yichao Zhang, Tim Fischer, Nils Wistoff, Luca Bertaccini, Thomas Benz, Luca Colagrande, Alfio Di Mauro, Andreas Kurth, Samuel Riedel, Noah Huetter, Gianmarco Ottavi, Zerun Jiang, Beat Muheim,

Frank K. Gurkaynak, Davide Rossi, and Luca Benini. Occamy: A 432-core, Multi-TFLOPs RISC-V-Based 2.5D Chiplet System for Ultra-Efficient (Mini-)Floating-Point Computation . *PULP Platform, ETH Zurich*, 2023.

[RAD78] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[Ram] Rambus. HBM3 Memory: Break Through to Greater Bandwidth.

[RCK+20] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S. Lee, and David Brooks. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference, 2020.

[RD24] United Kingdom Robert Dimond, System Architect ARM. Keynote: Chiplet standards: A new route to arm-based custom silicon. Technical report, ARM, 2024.

[RJV+15] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede. Modular hardware architecture for somewhat homomorphic function evaluation. In *Cryptographic Hardware and Embedded Systems - CHES*, 2015.

[RJV+18] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Transactions on Computers*, 2018.

[RKRB] Midia Reshadi, Ahmad Khademzadeh, Akram Reza, and Maryam Bahmani. A novel mesh architecture for on-chip networks.

[RLPD20] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[Sco17a] Michael Scott. A note on the implementation of the number theoretic transform. In *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*, pages 247–258. Springer, 2017.

[Sco17b] Michael Scott. A note on the implementation of the number theoretic transform. In *Cryptography and Coding - 16th IMA International Conference, IMACC 2017, Oxford, UK, December 12-14, 2017, Proceedings*, pages 247–258. Springer, 2017.

[Sco20] Michael Scott. Linear regression - insurance dataset, 2020.

[SCV+21] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Ross Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel S. Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: scaling deep-learning inference with chiplet-based architecture. *Commun. ACM*, 64(6):107–116, 2021.

[SFK+22] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. CraterLake: A hardware accelerator for efficient unbounded

computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 173–187, New York, NY, USA, 2022. Association for Computing Machinery.

[Sie14] Mark Siegesmund. Chip memory. Technical report, ScienceDirect, 2014.

[SRTJ+19] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 387–398, 2019.

[Syn23] Synopsys. How universal chiplet interconnect express changes soc design, 2023.

[SYY+23] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying CXL memory with genuine cxl-ready systems and devices. *CoRR*, abs/2303.15375, 2023.

[TCDM21] Zhanhong Tan, Hongyu Cai, Runpei Dong, and Kaisheng Ma. Nn-baton: DNN workload orchestration and chiplet granularity exploration for multi-chip accelerators. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*, pages 1013–1026. IEEE, 2021.

[TGF09] Thorlindur Thorolfsson, Kiran Gonsalves, and Paul D. Franzon. Design automation for a 3DIC FFT processor for synthetic aperture radar: a case study. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 51–56. ACM, 2009.

[TRG+20] Jonathan Takeshita, Dayane Reis, Ting Gong, Michael Niemier, X. Sharon Hu, and Taeho Jung. Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram. Cryptology ePrint Archive, Report 2020/1223, 2020. https://ia.cr/2020/1223.

[UCI23] UCIe. For the first time, ucie shares bandwidth speeds between chiplets, 2023.

[VGT+20] Pascal Vivet, Eric Guthmuller, Yvain Thonnart, Gaël Pillonnet, Guillaume Moritz, Ivan Miro-Panades, César Fuguet Tortolero, Jean Durupt, Christian Bernard, Didier Varreau, Julian J. H. Pontes, Sébastien Thuries, David Coriat, Michel Harrand, Denis Dutoit, Didier Lattard, Lucile Arnaud, Jean Charbonnier, Perceval Coudrain, Arnaud Garnier, Frédéric Berger, Alain Gueugnot, Alain Greiner, Quentin L. Meunier, Alexis Farcy, Alexandre Arriordaz, Séverine Cheramy, and Fabien Clermidy. 2.3 A 220gops 96-core processor with 6 chiplets 3d-stacked on an active interposer offering 0.6ns/mm latency, 3tb/s/mm^2 inter-chiplet interconnects and 156mw/mm^2@ 82%-peak-efficiency DC-DC converters. In *2020 IEEE International Solid- State Circuits Conference, ISSCC 2020, San Francisco, CA, USA, February 16-20, 2020*, pages 46–48. IEEE, 2020.

[VJHH23] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. HECO: fully homomorphic encryption compiler. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4715–4732. USENIX Association, 2023.

[WH13] Wei Wang and Xinming Huang. Fpga implementation of a large-number multiplier for fully homomorphic encryption. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2589–2592, 2013.

[WHEW14] Wei Wang, Xinming Huang, Niall Emmart, and Charles Weems. Vlsi design of a large-number multiplier for fully homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):1879–1887, 2014.

[XZH21] Guozhu Xin, Yifan Zhao, and Jun Han. A multi-layer parallel hardware architecture for homomorphic computation in machine learning. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[YCH22] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. Pipentt: A pipelined number theoretic transform architecture. *IEEE Trans. Circuits Syst. II Express Briefs*, 69(10):4068–4072, 2022.

[YLK+18] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natalie D. Enright Jerger, and Gabriel H. Loh. Modular Routing Design for Chiplet-Based Systems. In Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi, editors, *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 726–738. IEEE Computer Society, 2018.

[ZSB21] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro*, 41(2):36–42, 2021.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428. IEEE, 2021.